USENIX
ASSOCIATION

19th USENIX Symposium on Networked Systems Design and Implementation (NSDI '22)

Renton, WA, USA
April 4–6, 2022

Sponsored by

usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

In cooperation with
ACM SIGCOMM and ACM SIGOPS

NSDI '22 Sponsors

Platinum Sponsor

amazon

Diamond Sponsor

SIEMENS

Gold Sponsor

Meta

Silver Sponsors

ByteDance   FUTUREWEI Technologies   Google   intel.

Bronze Sponsors

IBM   Microsoft   TWO SIGMA   VMWARE UNIVERSITY RESEARCH FUND

Open Access Sponsor

King Abdullah University of Science and Technology

USENIX Supporters

USENIX Patrons

Amazon • Ethyca • Google • Meta
Microsoft • NetApp • Salesforce

USENIX Benefactors

AuriStor • Bloomberg • Discernible • Goldman Sachs • IBM
Shopify • Thinkst Canary • Transcend • Two Sigma

USENIX Partner

Blameless • Lightstep • Top10VPN

Open Access Supporter

Google

Open Access Publishing Partner

PeerJ

Proceedings of the
19th USENIX Symposium on
Networked Systems Design and Implementation

Conference Organizers

Program Co-Chairs

Amar Phanishayee, *Microsoft Research*

Vyas Sekar, *Carnegie Mellon University*

Program Committee

Sangeetha Abdu-Jyothi, *University of California, Irvine, and VMware Research*

Fadel Adib, *Massachusetts Institute of Technology*

Behnaz Arzani, *Microsoft Research*

Anirudh Badam, *Microsoft Research*

Mahesh Balakrishnan, *Facebook*

Aruna Balasubramanian, *Stony Brook University*

Hitesh Ballani, *Microsoft Research*

Sujata Banerjee, *VMware Research*

Theo Benson, *Brown University*

Matthew Caesar, *University of Illinois at Urbana–Champaign*

Vijay Chidambaram, *The University of Texas at Austin*

Asaf Cidon, *Columbia University*

Angela Demke Brown, *University of Toronto*

Fahad Dogar, *Tufts University*

Giulia Fanti, *Carnegie Mellon University*

Rodrigo Fonseca, *Microsoft Research*

Manya Ghobadi, *Massachusetts Institute of Technology*

Soudeh Ghorbani, *Johns Hopkins University*

Phillipa Gill, *Google*

Brighten Godfrey, *University of Illinois at Urbana–Champaign and VMware*

Shyam Gollakota, *University of Washington*

Ramesh Govindan, *University of Southern California*

Chuanxiong Guo, *ByteDance*

Andreas Haeberlen, *University of Pennsylvania*

Kurtis Heimerl, *University of Washington*

Wenjun Hu, *Yale University*

Kyle Jamieson, *Princeton University*

Junchen Jiang, *University of Chicago*

Anuj Kalia, *Microsoft Research*

Anurag Khandelwal, *Yale University*

Ana Klimovic, *ETH Zurich*

Dejan Kostic, *KTH Royal Institute of Technology*

Franck Le, *IBM Research*

Kate Lin, *National Chiao Tung University*

Zaoxing Alan Liu, *Boston University*

Jay Lorch, *Microsoft Research*

Jonathan Mace, *Max Planck Institute for Software Systems (MPI-SWS)*

Harsha Madhyastha, *University of Michigan*

Aurojit Panda, *New York University*

Kyoongsoo Park, *Korea Advanced Institute of Science and Technology (KAIST)*

Chunyi Peng, *Purdue University*

Ben Pfaff, *VMware Research*

George Porter, *University of California, San Diego*

Costin Raiciu, *University Politehnica of Bucharest and Correct Networks*

Robert Ricci, *University of Utah*

Michael Schapira, *The Hebrew University of Jerusalem*

Stefan Schmid, *Technische Universität Berlin and University of Vienna*

Brent Stephens, *University of Utah*

Laurent Vanbever, *ETH Zurich*

Shivaram Venkataraman, *University of Wisconsin—Madison*

David Walker, *Princeton University*

Jia Wang, *AT&T Labs*

Michael Wei, *VMware Research*

John Wilkes, *Google*

Xiaowei Yang, *Duke University*

Minlan Yu, *Harvard University*

Ellen Zegura, *Georgia Institute of Technology*

Ying Zhang, *Facebook*

Yiying Zhang, *University of California, San Diego*

Ben Zhao, *University of Chicago*

Wenting Zheng, *Carnegie Mellon University*

Lin Zhong, *Yale University*

Danyang Zhuo, *Duke University*

Steering Committee

Aditya Akella, *University of Wisconsin–Madison*

Sujata Banerjee, *VMware Research*

Ranjita Bhagwan, *Microsoft Research India*

Casey Henderson, *USENIX Association*

Jon Howell, *VMware Research*

Arvind Krishnamurthy, *University of Washington*

Jay Lorch, *Microsoft Research*

James Mickens, *Harvard University*

Jeff Mogul, *Google*

George Porter, *University of California, San Diego*

Timothy Roscoe, *ETH Zurich*

Srinivasan Seshan, *Carnegie Mellon University*

Renata Teixeira, *Netflix*

Minlan Yu, *Harvard University*

External Reviewers

Anubhavnidhi "Archie" Abhashkumar

Rachit Agarwal

Fawad Ahmad

Lixiang Ao

Rodrigo Bruno

Ang Chen

Italo Cunha

Weiqi Feng

Bryan Ford

Jiaqi Gao

Rajrup Ghosh

Junzhi Gong

Arpit Gupta

Indranil Gupta

Dongsu Han

Yitao Hu

Ryan Huang

Keon Jang

Weifan Jiang

Swarun Kumar

Chonlam Lao

Minghao Li

Devon Loehr

Pooria Namyar

Dave Oran

Dan Pei

Sivaram Ramanathan

Christopher Rossbach

Siddhartha Sen

Srini Seshan

Srinath Setty

Rob Sherwood

Alex Snoeren

Jiri Srba

Srikanth Sundaresan

Francois Taiani

Matteo Varvello

Yongqiang Xiong

Zhiying Xu

Francis Yan

Michelle X. Yeo

Irene Zhang

Mingyang Zhang

Yang Zhou

Noa Zilberman

Message from the
NSDI '22 Program Co-Chairs

Welcome to NSDI 2022!

We live in unprecedented times. We have been through waves of multiple covid variants, parents dealing with the uncertainty of school schedules, rapid scientific breakthroughs that resulted in effective vaccines, mass vaccination drives, and war in parts of the world that threatens dislodging the lives of millions of people. It is difficult to reason about the importance of our work in such turbulent times. But despite common as well as uniquely individual challenges, our community marches on, building on lessons we have learnt on operating during the pandemic.

NSDI '22 received a record number of submissions—396 papers in total: 104 in spring and 292 for the fall deadline. A total of 78 papers were accepted for an acceptance rate of 19.7%—the highest we have seen in a while. Papers were reviewed by a program committee of 65 experts from both academia and industry. One-shot revisions, first introduced to NSDI in 2019, have proved to be quite successful and we continue this practice.

We sincerely thank our reviewers who provided thoughtful feedback to our authors, including those who re-reviewed one-shot revisions from the prior NSDI edition (NSDI '21 Fall) and our expert external reviewers. We also want to thank our stand-in PC chairs (for chair-conflict papers) who helped out selflessly performing tasks that they had not necessarily signed up for when they agreed to be on our PC: Ellen Zegura, Mahesh Balakishnan, and Ben Y. Zhao. We thank George Porter and Harsha V. Madhyastha for going above and beyond the call of duty—and at short notice—to carefully read and help us select the best paper award winners this year. We are also very grateful to prior NSDI chairs Arvind Krishnamurthy, Jay Lorch, James Mickens, and Renata Teixeira for sharing their best practices with us. And we thank student volunteers Brian Singer and Milind Srivatsava for helping us run a smooth Fall PC meeting over Zoom. Finally, we also thank the paper authors; your submissions are what make NSDI such a great venue, and we hope that you will enjoy the conference program.

We are trying two new experiments this year. First, NSDI will be held as a hybrid event. We are excited that the program will be held in-person, but we recognize that there are many authors and attendees who will only be able to attend virtually as the threat of a new Covid variant looms large in many parts of the world. Second, for both safety (to avoid packing all attendees in a single room), as well as providing the large number of accepted papers with ample time to present their ideas, we are experimenting with a dual-track format. While changes to well-established ways of doing things make us a little anxious, we could not be more excited that USENIX is the organization shepherding us through these changes.

Which brings us to thanking one of the most important groups that has helped us: USENIX. We'd like to thank all of the USENIX staff who helped us to organize this year's conference right from the get go: from configuring the HotCRP server to dealing with camera-ready production of both papers and talks (and they had to do this twice for the spring and fall deadline), the USENIX staff provided invaluable advice and flawless execution. We are certain we will miss many names we ought to thank, and in some cases because we magically saw the result of your work but never got to know that you did it. Our heartfelt thanks to Casey Henderson (for patiently helping us on so many different dimensions), Olivia Vernetti, Camille Mulligan, Arnold Gatilao, Jasmine Murcia, Jessica Kim, Julia Hendrickson, Liz Markel, Sarah TerHune, and the rest of the USENIX team. You are a magical team, and we as a community are lucky to have such dedicated, caring, and supremely competent USENIX staff. We would be lost in the wilderness without you.

Stay safe and healthy, and enjoy NSDI in whichever format you choose to attend it!

Amar Phanishayee, *Microsoft Research*
Vyas Sekar, *Carnegie Mellon University*
NSDI '22 Program Co-Chairs

19th USENIX Symposium on Networked Systems Design and Implementation (NSDI '22)

April 4–6, 2022

Renton, WA, USA

Monday, April 4

Cluster Resource Management

Transport Layer - Part 1

Video Streaming

Programmable Switches - Part 1

Security and Privacy

Network Troubleshooting and Debugging

Operational Track - Part 1

Wireless - Part 1

Tuesday, April 5

Reliable Distributed Systems

Raising the Bar for Programmable Hardware

Testing and Verification

Programmable Switches - Part 2

Wednesday, April 6

Operational Track - Part 2

Edge IoT Applications

Cloud Scale Services

ISPs and CDNs

Cloud Scale Resource Management

Data Center Network Infrastructure

Multitenancy

Software Switching and Beyond

Efficient Scheduling Policies for Microsecond-Scale Tasks

Sarah McClure*, Amy Ousterhout*, Scott Shenker*†, Sylvia Ratnasamy*
*UC Berkeley †ICSI

Abstract

Datacenter operators today strive to support microsecond-latency applications while also using their limited CPU resources as efficiently as possible. To achieve this, several recent systems allow multiple applications to run on the same server, granting each a dedicated set of cores and reallocating cores across applications over time as load varies. Unfortunately, many of these systems do a poor job of navigating the tradeoff between latency and efficiency, sacrificing one or both, especially when handling tasks as short as 1 μs.

While the implementations of these systems (threading libraries, network stacks, etc.) have been heavily optimized, the policy choices that they make have received less scrutiny. Most systems implement a single choice of policy for *allocating cores* across applications and for *load-balancing* tasks across cores within an application. In this paper, we use simulations to compare these different policy options and explore which yield the best combination of latency and efficiency. We conclude that work stealing performs best among load-balancing policies, multiple policies can perform well for core allocations, and, surprisingly, static core allocations often outperform reallocation with small tasks. We implement the best-performing policy choices by building on Caladan, an existing core-allocating system, and demonstrate that they can yield efficiency improvements of up to 13-22% without degrading (median or tail) latency.

1 Introduction

Modern datacenter applications often involve many short Remote Procedure Calls (RPCs) to other servers. These RPCs allow applications with large memory footprints to access memory on other servers [2, 49, 51, 62, 69], enable applications to leverage large amounts of compute over short timescales [6, 25, 46], and provide replication and consensus [58]. The service times of these tasks grow ever smaller, and today are often a single microsecond or less [10, 34].

Tasks with short service times are particularly vulnerable to latency inflation; even small overheads can increase the latency of a 1 μs task by an order of magnitude [10]. This is problematic for today's applications, which depend on low latency both at the median and at the tail of the distribution (e.g., 99% latency) [5, 19]. As a result, researchers have proposed many techniques to reduce the overheads of handling these short tasks. These systems improve software with low-latency network stacks and better load balancing (DPDK [1], ZygOS [66], Shinjuku [36], eRPC [38], etc.) or propose new hardware to deliver packets to cores more quickly (RPC-

Valet [18], NeBuLa [74], NanoPU [34], Cerebros [65]). They offer tail latencies of a few dozen microseconds with existing hardware [26, 38] or several microseconds with new hardware [34].

However, as Moore's Law slows [23], datacenter operators are increasingly concerned not just with providing low latency but also with achieving high CPU efficiency [79]. To do so, they pack multiple applications on the same server so that background applications can use any CPU cycles not used by latency-sensitive applications, as their load varies over time [11, 35, 80]. Several recent research systems enable this deployment model by allocating a set of cores to each application and then reallocating cores across applications as load changes (e.g., IX [12], PerfISO [35], Arachne [67], Shenango [60], Caladan [26], and Fred [40]). These systems walk a delicate tightrope, attempting to make spare cycles available for batch applications without harming the latency or throughput of latency-sensitive applications. Thus researchers have heavily optimized these systems' implementations, squeezing spare CPU cycles and extraneous cache misses out of their network stacks, threading libraries, and core-allocation mechanisms.

While there have been significant advances in these mechanisms, less effort has gone into studying the *policies* that these core-reallocating systems implement. Each system implements two main policies: (1) a policy for **load-balancing** tasks across cores within an application and (2) a policy for when to **reallocate cores** from one application to another. There are many possible choices for each policy: popular load-balancing policies include work stealing [14], work shedding, and steering tasks to less-loaded cores when they are first enqueued [55] while core-allocation policies may be based on queueing delay [12, 26, 60], the arrival of new tasks [40], or CPU utilization [35, 67]. And yet, each system typically implements a single choice of load-balancing and core-allocation policy, providing little clarity about how different policies compare.

Unfortunately, as we will show (§2), these policy choices can contribute to suboptimal performance, with existing systems sacrificing significant CPU efficiency in order to maintain low latency, especially with short tasks. The root of the problem is that as task durations shrink from 100 μs to 1 μs, the overheads of balancing tasks or reallocating cores (e.g., a 50 ns cache miss to probe state on a different core) become relatively more significant, and inefficient policies become much more costly. In this paper, we focus on these policies and ask: *what load-balancing and core-allocation policies*

yield the best combination of latency (median and tail) and CPU efficiency for microsecond-scale tasks?

We focus on the *combination* of latency and efficiency because while ideally we would like to optimize both, there is an inherent tradeoff between the two. For example, allocating infinite cores could achieve optimal latency at the cost of terrible efficiency, while allocating a single core could achieve good (but perhaps not optimal) efficiency, but potentially high latency. The best one can hope for is to operate on the Pareto frontier of latency and efficiency; i.e., a point where it is not possible to improve one quantity without harming the other.

To compare policies fairly and independently from any specific implementation, we turn to simulations (§4). We use measurements of real systems to estimate the overheads of balancing tasks across cores within an application and of re-allocating cores across applications. We then model simple versions of common load-balancing and core-allocation policies, and simulate them using our estimated overheads. We use these simulations to conduct an extensive factor analysis, teasing apart the impact of load-balancing policies and core-allocation policies on both latency and efficiency. From this analysis, we glean three key insights:

First, assuming commodity NIC hardware, *work stealing* is the load-balancing policy that yields the best latency and CPU efficiency and forms the Pareto frontier. We find that this conclusion is remarkably robust across different average service times, service time distributions, numbers of cores, latency metrics (e.g., median vs. 99%), whether cores are dynamically reallocated or statically partitioned, and how much overhead load-balancing a task entails.

Second, in contrast, our analysis of core-allocation policies shows that multiple policies can perform similarly well (though some policies perform significantly worse). We find that *revoking cores proactively, rather than waiting until they go idle to yield them to another application, makes it easier to achieve good efficiency with small tasks, especially with many cores*. We identify two policies (based on average queueing delay and average CPU utilization) that fit this criteria, perform well, and can be configured to make different tradeoffs along the Pareto frontier; two other policies used in current systems yielded worse latency, CPU efficiency, or both.

Third, even with the best core-allocation policies, if the average load is fixed, *with small tasks it is difficult to achieve better performance by reallocating cores than by allocating a fixed number of cores*. For our request patterns (modeled with exponentially-distributed inter-arrival times), reallocating cores in response to transient bursts does not improve latency (median or tail) relative to statically allocating the same average number of cores. Thus the main benefit of re-allocating cores over short timescales with short tasks is the ability to quickly adapt to changes in *average* load. In contrast, when average task service times are longer—several microseconds or more—we find that reallocating cores does improve performance even with constant average load.

(a) Arachne. **(b)** Caladan.
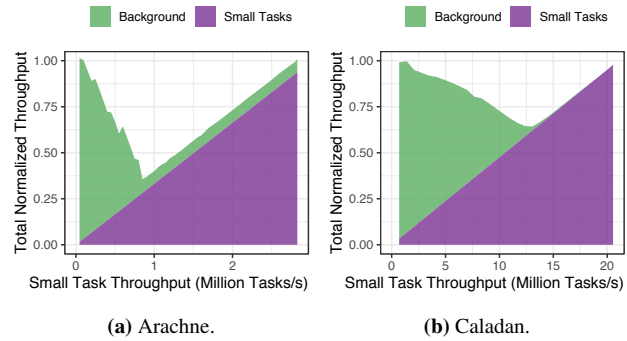
Figure 1: Total useful work done by two colocated applications—one background, the other handling small (about 1 μs) memcached tasks—as we vary memcached's load (for two existing systems).

From this factor analysis we conclude that barring technology changes (e.g., commercialization of recently proposed NIC hardware [18, 34, 65, 74]), for low latency and high CPU efficiency, work stealing is the best load-balancing policy, and our two new core-allocation policies based on average delay or average utilization (we refer to these policies as "delay range" and "utilization range") perform best. We implement these policies in a real system by extending Caladan [26], a state-of-the-art system for reallocating cores which already supports work stealing. We demonstrate that when running memcached, a key-value store, delay range and utilization range can save up to 13-22% of cores relative to Shenango's and Caladan's core-allocation policies, without degrading median or tail latency (§6).

2 Motivation

To demonstrate the inefficiencies of existing systems when handling short tasks, we conduct an experiment in which we run two applications on a server: a latency-sensitive application that handles short tasks and a background application that consumes all extra CPU cycles. We use memcached [49], a key-value store with service times of about 1 μs, as our latency-sensitive application. We vary the offered rate of memcached tasks and measure how much useful application-level work each application completes. We perform this experiment with two existing systems: Arachne [67] and Caladan [26].

Both systems yield latency improvements: Arachne's 99% latency improves on that of Linux by hundreds of microseconds, while Caladan reduces this further, due partially to replacing Linux's network stack with kernel bypass. However, in their efforts to provide low latency for the small tasks, these systems waste significant CPU resources. Figure 1 shows the total throughput achieved by each system, normalized by the maximum throughput the application can achieve when running alone on the configured set of cores (16 for Arachne and 32 for Caladan). Thus at the lowest and highest loads (where only one of the applications is running[1]), both systems are

---

[1]Arachne dedicates one core to each application, so its background throughput never reaches zero.

at their highest possible efficiency, achieving a total normalized throughput of 1.0. Ideally, the total throughput of both applications would remain at 1.0 as the small task load varies. However, at moderate loads, both systems suffer significant efficiency losses, wasting up to 64% or 36% of their cores, with Arachne and Caladan, respectively. This inefficiency is not exclusively bad; the excess cycles can be used to handle small tasks sooner, lowering latency.

From these results, it is clear that these systems are able to multiplex cores between applications, but they are extremely inefficient while doing so. When handling longer tasks (e.g., 10 μs or 100 μs), these systems become much more efficient. This begs the question: what is responsible for these efficiency losses with short tasks? These systems differ along many different dimensions: their core-allocation policies, their load-balancing policies, their threading libraries, and whether they use the Linux network stack (Arachne) or kernel-bypass (Caladan). The latter implementation aspects can contribute significantly, but they have been studied extensively by prior work. We focus instead on the policy aspects and seek to understand which load-balancing and core-allocation policies yield the best performance for small tasks.

3 Design Space of Policies

If reallocating cores across applications and load balancing tasks between cores incurred no overhead (i.e., they could be done instantaneously), the optimal policies would be: (1) immediately grant an application a new core whenever a task arrives and yield the core when the task completes and (2) steer each newly arrived task to its newly granted core. With these policies, CPU usage would exactly match the time spent on tasks (100% efficient) and if an additional core was always available then tasks would never queue (zero added latency).

These idealized policies are sufficient with long task service times (e.g., 100 μs or more), because the overheads of load balancing and core reallocation are relatively small (§4.3). However, with tasks as short as a single microsecond, load-balancing and core-allocation overheads become significant and we can no longer afford to perform both a core-allocation and a load-balancing action for every task that arrives; doing so wastes considerable CPU resources. For good performance with short tasks we must consider other policies. The key difference between distinct policies is when they choose to incur overheads (e.g., when a task arrives vs. when a queue builds up), and these choices determine their latency and CPU efficiency. Thus finding the best load-balancing and core-allocation policies amounts to asking the question: *given that load balancing and core allocation incur overheads, how should we spend those overheads most effectively?*

3.1 Setting and Assumptions

While exploring different policies, we make several assumptions about our setting (illustrated in Figure 2). We assume that each server runs one or more *applications*, where each ap-
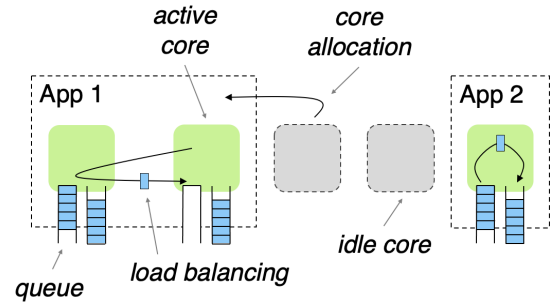
Figure 2: Applications use *load balancing* to balance tasks across cores and *core allocations* to adjust the number of cores available to each application.

plication is either a batch application that seeks high throughput and is latency-insensitive or a latency-sensitive application that handles short *tasks*. Each application is always allocated a specific number of cores; when an application yields a core, the core will be granted to another application if possible.

Tasks can either arrive from external sources (e.g., a packet arrives over the network or a storage operation completes) or be created by the local CPU (e.g., a thread spawns a new thread). We focus on settings with commodity NICs that spray packets randomly over available cores (e.g., with RSS [3]), though we also discuss how performance could change with recent proposals for new NIC hardware with advanced steering capabilities (§4.2.1). Unless specified otherwise, we assume that each core maintains its own queue(s) of tasks and that tasks are not intentionally re-ordered (cores handle them in FIFO order). We assume no preemption of running tasks and no *a priori* knowledge of how long each task will take to run.

3.2 Policies

In this section, we summarize the main policies used for load balancing and core allocation today and describe when each incurs overheads; these are the policies we evaluate in our factor analysis (§4). The list is not exhaustive but rather an attempt to cover the main classes of existing policies as well as the theoretically optimal policies.

3.2.1 Load-Balancing Policies

Load-balancing policies can perform load balancing either when a task arrives or once it has already been queued. The latter category can be further divided based on what triggers load balancing (either a lack of tasks for a core or a core with too many tasks). We begin by describing a theoretical optimum, and then describe four practical policies that fall into these categories. Note that these policies are not necessarily mutually exclusive.

Single queue. With no overheads, the theoretically ideal load-balancing policy places all tasks in a single shared queue. However, this approach limits throughput in practice due to contention for the single queue. Shinjku [36] and RAM-Cloud [62] take this approach; Shinjuku can support only

System	Load-balancing Policy	Core-allocation Policy	
		Trigger for Adding a Core	Trigger for Revoking a Core
IX [12]	none	packet queueing delay	low CPU utilization
Arachne [67]	choice on enqueue with power-of-two choices [55]	number of runnable threads	low CPU utilization
Shenango [60], Caladan [26]	work stealing	max queueing delay of threads or packets	failure to work-steal
Fred [40]	steering on arrival or work-stealing once all cores are allocated	task arrives	task completes
Go [76]	work stealing	task arrives and no cores work stealing	failure to work-steal

Table 1: Load-balancing and core-allocation policies used by existing systems. Core-allocation policies are highlighted to indicate whether they rely on queueing , utilization , task arrival , or failure to find work .

about 5 million requests per second with a single queue.

No load balancing. Without load balancing, tasks are handled by the core they first arrive at, such as the core that spawns a thread or the core a packet or storage completion is steered to by hardware [12, 42, 64]. This approach incurs no load-balancing overheads.

Enqueue choice. Enqueue choice policies make a load-balancing decision about which core to assign a task to when the task is first created; tasks cannot be moved later. Existing systems commonly use "power of two choices" [55] to enqueue a task to the less-loaded of two randomly sampled cores [33, 67, 81]. When a task is first created, the creating core incurs overhead to sample queues on other cores (which can be done in parallel for small numbers of sampled cores) and to enqueue the task to the chosen core.[2]

Work stealing. When a core is idle, it searches for a core that has queued work, and then steals half the tasks from that core and moves them to its own queue [14]. This approach is used by the Go runtime [76], several multithreading platforms [9, 16, 17, 45, 47, 59, 68], and many research systems [26, 40, 44, 50, 60, 66, 81]. It incurs overhead to check other cores for queued work and move work to its local queue.

Work shedding. With work shedding, overloaded cores can shed load to other cores or request that other cores take some of their load. This has been considered by several theoretical papers [22, 73, 77] and for load-balancing systems in a variety of contexts [56, 78]. We consider a work-shedding policy in which a core that has had tasks queued for longer than a specified threshold selects a random core and indicates that it is overloaded. That core will then respond by stealing half of the overloaded core's tasks; this is the primary source of overhead for this policy.

3.2.2 Core-Allocation Policies

All core-allocation policies incur overhead in the same way: by adding or revoking a core. Their overheads are primarily determined by how often they reallocate cores and consist of both the latency until a core is available after a reallocation decision is made and the CPU cycles that cannot be used productively while a core is being reallocated. The performance of each policy is determined by how effective the signals are that it uses to trigger core reallocations. Most policies make

core-allocation decisions at fixed time intervals (e.g., every 5 μs [60]), though some are triggered by other conditions.

We cannot easily model or compute an optimal core-allocation policy, i.e., one that achieves the optimal tail latency for a given CPU efficiency or vice versa. This is because finding the optimal tail latency for a given CPU usage bound or vice versa is NP-hard assuming a finite number of cores and non-constant service times; this can be shown by a reduction from the multiprocessor scheduling problem (see Appendix A.1). We now list the core-allocation policies we consider.

Static. With static core allocations, the number of cores allocated to each application cannot change over time, as in several research systems [42, 64, 66]. This incurs no overhead for core reallocations. However, each application must be provisioned with enough cores for peak load, wasting significant CPU resources as load varies over time, which is typical of datacenter workloads [11, 35].

Per-task. Systems such as Fred [40] with per-task core allocations grant a core to an application every time a task arrives. This incurs the overhead of a core allocation for each task, except when all cores are in use.[3]

Queueing-based. Policies based on queueing delay grant an application an additional core if the queueing—as measured by either the number or delay of threads, packets, or storage completions—exceeds a certain threshold. These policies vary in whether they trigger based on the maximum queueing across cores [26, 60] or use an average [12, 67].

CPU utilization-based. Utilization-based policies add or revoke cores based on the number of idle cores [35] or the average fraction of time cores spend working on tasks (as opposed to sitting idle or busy-spinning) [12, 67].

Failure to find work. In some systems, an application will yield a core when the core is unable to find any tasks to work on. This can happen when a core fails to find another core with queued work to steal from [26, 60, 76] or when it finishes its current task, with a per-task core-allocation policy [40].

3.3 Overheads

Both load balancing and core allocation entail overheads; in this section we discuss the magnitude of these overheads in typical systems today.

---

[2]Note that "no load balancing" is a special case of enqueue choice in which there is only one choice and no overhead.

[3]Once all cores are allocated to an application, Fred places additional arriving tasks in per-core queues and cores use work stealing to find them.

Load-balancing overheads. Load-balancing overheads can be impacted by several factors: the CPU architecture (how long does it take to handle a cache miss? how many cache misses can be outstanding simultaneously?), the workload (how often is load-balancing state cached locally vs. modified on remote cores?), and speculative execution (how successfully can the CPU overlap cache misses with other instructions via speculative execution?). Despite these factors, we attempt to estimate the overheads of different load-balancing policies and in Section §4.2.1 we demonstrate that our conclusions about the relative performance of different policies are unlikely to change with different overheads.

Because load balancing requires communication between cores, its overhead arises primarily from cache misses while retrieving cache lines from the L2 cache of another core. Depending on the CPU microarchitecture, one such cache miss can cost between 30 ns (Intel Haswell) and 200 ns (Xeon Phi) [72]. A load-balancing operation moves state from one core to another; this typically entails about three cache misses to read a remote cache line, invalidate it so that it can be written in the local cache, and then a third cache miss when the remote core reads the modified cache line [67]. The overhead incurred by the core performing the load balancing will then be about two cache misses, or 60-400 ns.[4] For comparison, we measured that Caladan [26] takes about 120 ns on average to check via work stealing if another core has stealable work (in the form of queued packets, threads, or timers).

Note that a single core can typically have up to about 10 cache misses outstanding at once [24] (we confirmed through a microbenchmark [48] that this seems to be about 10-12 for our Intel Skylake servers). This enables small numbers of independent cache misses (such as those to sample the load on two different cores) to incur in parallel.

Core-allocation overheads. The latency for a core allocation to complete varies depending on the mechanism used to reallocate the core. At a bare minimum, reallocating a core requires an inter-processor interrupt (IPI) from the core that makes the reallocation decision to the core that will be reallocated to a different application; this takes about 1993 cycles or roughly 1 μs [36]. Existing systems report slightly higher core-allocation latencies, varying from 2.2 μs to reallocate an idle core or 7.4 μs to reallocate a busy core in Shenango [61] to 29 μs to reallocate a core in Arachne [67].

4 Factor Analysis

In this section, we perform a factor analysis to determine the relative performance of the load-balancing and core-allocation policies defined in §3.2. We cannot effectively compare different policies by comparing existing systems that implement them (e.g., Caladan vs. Arachne), because these systems differ in many aspects besides their policies (threading libraries, net-

work stacks, etc.). Even comparing different policies within a single implemented system can be challenging, because the optimal system design may vary depending on the policy. For example, systems may use different locking mechanisms to protect thread queues depending on whether only the local core can enqueue to them (as in work stealing) or if remote cores can also enqueue to them (as in enqueue choice). Thus, to decouple the behavior of the policies from the behavior of the systems that they are implemented in, we use simulations.

Our simulations rely on several parameters which define both the workload and assumptions about the possible underlying system. We find that our conclusions are quite robust to variations in these parameters, and therefore may be applicable to a wide variety of implementations and workloads. We have made the source code for our simulations available at https://github.com/smcclure20/scheduling-policies-sim.

4.1 Simulation Methodology

While our focus is on policy choices rather than implementation details, we do seek to model realistic overheads for cross-core communication and for allocating cores to applications. In order to fairly compare different policies, we use consistent values for each overhead, based on the overheads measured above (§3.3). We model the cross-core communication generally required for load balancing as taking 100 ns. We model the core-allocation overheads (both latency to allocate a core and wasted CPU cycles) as 5 μs per core allocation. In §4.2.1, we will consider some different values for load-balancing overheads, though varying them by even 100% does not have a profound impact on our results. We discuss the implications of varying core-allocation overheads in §4.3.

Our overall model assumes that each core has a single local queue (i.e., no distinction between packet and thread queues) and that tasks arrive randomly at the queues of allocated cores. This is representative of a NIC randomly steering tasks to cores or of running threads randomly spawning an additional thread. Our simulator models each of the general policy approaches outlined in §3.2, with specific implementation choices made based on real system implementations whenever possible. We acknowledge that our model is a simplified view of these systems, but we found that the general trends of latency and efficiency are consistent between simulations and experiments, for the systems we evaluated (§6). Our simulator does not support preemption but could be extended to model systems which do [20, 36, 82]. We now describe the specific load-balancing and core-allocation policies that we simulate.

Load-balancing policies. We model no overheads for the idealized *single-queue* policy or for the *no load balancing* policy. For *enqueue choice*, when a task arrives, the core at which it arrives incurs the 100 ns overhead to move the task to its destination queue (the shortest queue from two randomly

---

[4]This is an approximation; the exact overhead will depend on application behavior.

sampled options).[5] When *work stealing* is enabled and a core does not have any work in its local queue, it begins iterating through the other cores, checking if there is available work to steal. Each check of a remote queue incurs the 100 ns overhead, as does the act of stealing any found tasks. With *work shedding*, each core checks if its queue's current queueing delay is higher than the configured threshold after each task it finishes. If so, it selects a random core to notify or "flag." The remote core will check for flags between each of its tasks, respond (if a flag is present) by stealing tasks from the overloaded queue so that the two queue lengths are balanced, and incur the 100 ns overhead.

Core-allocation policies. Our *per-task* policy (based on Fred [40]) immediately grants a new core to an application if one is available in the system whenever a new task arrives. The core at which the task is initially randomly placed pays a 100 ns overhead to place the task at the new core. When a core finishes a task, it checks if there are more queued tasks in the system than available cores and yields if there are not.

The remaining core-allocation policies make decisions at fixed time intervals. To model *Shenango* [60] and *Caladan* [26], at the end of every core-allocation interval, the simulation determines the maximum queueing delay across cores within an application. If it exceeds a specified threshold (typically the length of the interval itself), the simulation grants an additional core to that application. An application yields a core if the core attempts to work steal from every other core in the application and fails to find any tasks to steal. Shenango and Caladan have very similar policies; the main distinguishing factor in our model is the difference in their interval/threshold values (Table 2).

We also design and simulate two new core-allocation policies. First, we design a queueing-based policy called *delay range* which attempts to maintain a specified average queueing delay across all cores within an application. Every core-allocation interval (every 5 μs), the simulation checks the average queueing delay. If it is below the specified lower bound, a core is revoked; if it is above the upper bound, a core is added. Similarly, with our *utilization range* policy, a core is added or removed whenever the average CPU utilization over the past interval (fraction of time spent handling tasks) falls outside the specified range.

There are three notable aspects of core-allocation systems that we do not model. First, some systems dedicate a scheduler core to making core-allocation decisions and initiating core allocations [26, 60, 67] while others have application cores perform these tasks in a distributed way [40, 76]. We do not model these distinctions and assume that all work for initiating core reallocations could be offloaded to a separate dedicated core. Second, we do not model the overheads incurred by applications measuring and exposing statistics to the dedicated core; in practice these overheads are small and

---

[5]We assume the options may be checked in parallel as explained in §3.3.

Parameter	Default Value
Work shedding delay threshold	2 μs
Enqueue choices	2
Utilization range	75-95%
Delay range	0.5-1 μs
Shenango max queueing threshold	5 μs
Caladan max queueing threshold	10 μs

Table 2: Canonical configuration parameters.

simply require application cores to write a small amount of state (e.g., timestamp when a task was queued) to shared memory. Third, we do not model the caching implications of reassigning a core from one application to another.

Configuration. Each policy has its own unique parameters. Unless stated otherwise, we use the default parameter values shown in Table 2. We chose these specific values based on the best overall performance seen for each policy, though we will discuss the implications of configurability throughout this section.

In all of our simulations, we use a canonical configuration of 32 cores, exponentially-distributed service times with an average of 1 μs, Poisson arrivals, and an offered load that occupies 50% of the cores on average. Experiments below will vary many of these dimensions independently, but we will use this configuration by default. To contextualize the policy overheads described above, with the average task time set at 1 μs, the overhead for load balancing is 10% of average task time while the overhead of core reallocation is 500%.

4.2 Load Balancing

To understand how load-balancing policies impact performance, we first evaluate different load-balancing policies in a setting where cores are statically allocated (cores are never reallocated) (§4.2.1), and then evaluate whether core reallocations impact these findings (§4.2.2).

4.2.1 With Static Core Allocations

Individual policies. We first evaluate each load-balancing policy independent of any particular core-allocation policy by running each experiment with a fixed number of cores. This allows us to determine the relative performance of each approach when given the same number of total CPU cycles, since a given allocation policy will make different allocation decisions depending on the behavior of the specific load-balancing scheme, even under the same traffic. By decoupling the two, we can determine which end-to-end effects are due specifically to load-balancing policies.

Figure 3 shows the tail and median latencies (y-axis) of different load-balancing policies as we vary the number of statically-allocated cores (shown on the x-axis as a fraction of the total possible), while offering an average load of 50%. Each curve corresponds to a load-balancing policy with 100 ns overheads, while the shaded regions vary this from 0 ns to 200 ns. In general, approaches that operate lower and to the left in this graph are preferable. We will discuss the JBSQ

(a) 99.9% latency.

(b) Median latency.
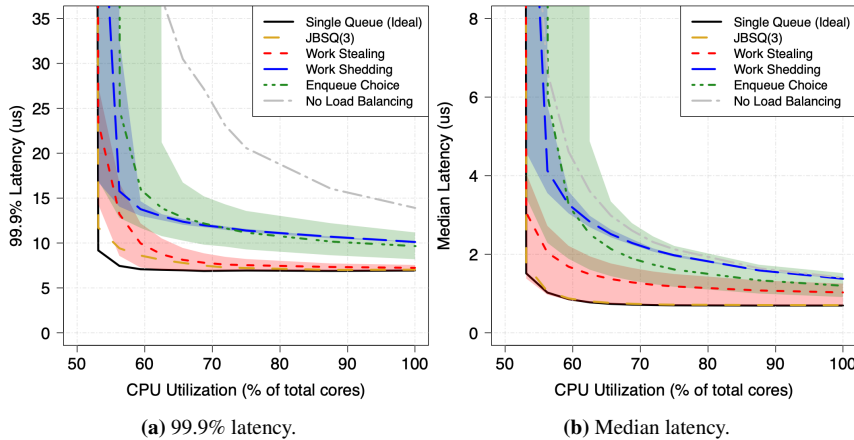
Figure 3: Performance of each load-balancing policy with different numbers of statically allocated cores. Shaded regions cover overheads 0-20% of average task time. The line in each region shows the canonical case of 10% load-balancing overheads (100 ns).

Figure 4: Latency curves for combinations of work stealing with other load-balancing policies, with static core allocations.

curve later.

Finding 1: *With static core allocations, work stealing achieves better latency (at the median and tail) for a given efficiency (number of allocated cores) than work shedding or enqueue choice.*

While all load-balancing policies yield significant improvements over no load balancing, work stealing consistently has significantly lower median and tail latency for the same number of statically-allocated cores than the other approaches; work stealing Pareto-dominates enqueue choice and work shedding. The relative performance between enqueue choice and work shedding is less consistent and varies depending on system and workload parameters such as the number of allocated cores, latency percentile, and service time distribution (Appendix A.2.2).

The enqueue choice curve is consistent with the well-known "power-of-two choices" result [55], showing that two choices of queues is much better than one (the "No Load Balancing" curve). This is particularly true when there is no overhead (as modeled in [55]) which is demonstrated by the lower bound of enqueue choice's shaded region in Figure 3. Despite this, enqueue choice still performs worse than work stealing. Further measurements revealed that this is due to three main limitations: (1) per-task load-balancing overheads that cap the possible throughput and add latency to all tasks, (2) a limited number of queue choices, and (3) placement based on number of queued tasks rather than the sum of service times of queued tasks. Overall, (2) and (3) can result in periods of load imbalance in which tasks are queued and cores are idle, but there is no way for the idle cores to assist with those "stranded" tasks. Choosing by the sum of the service times in the queue [30, 31] or increasing the number of choices can improve tail latency, though these are not always practical, and reducing the overheads to 0 provided a bigger performance benefit than either of those changes individually.

The tail latency gap between work stealing and work shed-

ding can be explained by the steps necessary to move a task that ends up contributing to tail latency to the core that ultimately handles it. With work shedding, for a task at an overloaded core, time is spent waiting to cross the signalling threshold, waiting for the core to complete its current task and raise a flag, and waiting for the remote core to respond. In work stealing, tasks simply wait until a work-stealing core checks their queue; the latency of this depends primarily on the number of excess cores. With a work-shedding queueing threshold of 2 μs we found that on average tasks that were shed spent 3.1-4.5 μs queued on cores other than the one that ultimately handled the task, compared to 0.3-1.4 μs with work stealing. Most tasks at the tail are stolen at least once, explaining the corresponding gap between the two in tail latency. Lowering the queueing threshold only yields marginal improvements, because at higher loads most cores will always have a flag pending. In addition, without preemption, tasks still incur delays from the other two steps described above.
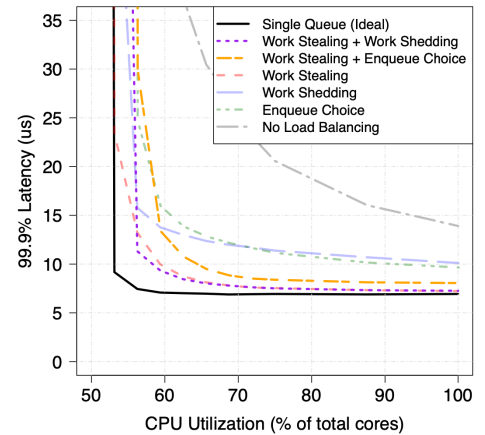
We now investigate the robustness of these results to changes in overheads in case our overhead estimates are not representative of certain underlying hardware or better technology arises in the future. Note that this does not apply to single queue simulations or those with no load balancing as they have no overheads. By looking at the upper or lower bounds of the shaded regions in Figure 3, we see that for the same overhead, work stealing consistently outperforms the other approaches. Even if inter-core communication was free for enqueue choice and work shedding, work stealing with 200 ns overheads outperforms for most numbers of allocated cores. Further, work stealing consistently achieves the best performance even if we model the load-balancing overhead as 400 ns, the upper bound of our estimate from §3.3.

Work stealing's superior performance is robust across different latency percentiles (median to 99.9%) (Figure 3), average service times (*e.g.,* 1, 10, 100 μs) (Figure 5), numbers of cores (Figure 6), loads (Appendiex A.2.1), and ser-

vice time distributions (exponential, constant, bimodal) (Appendix A.2.2). For all service time distributions evaluated, the ordering of the static curves remained the same as in exponential distributions shown. Though, when service times are constant, the specific choice of load-balancing policy has less overall impact on system performance.

Combining policies. Notably, these load-balancing approaches are not mutually exclusive. Since each policy takes effect at a different time in the handling of a task, work stealing can take advantage of extra cycles while work shedding addresses excessively loaded cores or enqueue choice proactively tries to balance queues. Accordingly, we simulated work stealing combined with each other approach with static core allocations.

Finding 2: *With static core allocations, adding shedding on top of work stealing provides some latency benefit (primarily at the tail) while adding enqueue choice to work stealing makes performance unchanged or worse.*

This is demonstrated in Figure 4. We see that adding enqueue choice to a system that already employs work stealing does not improve performance. There are two reasons for this, depending on what efficiency (x-axis) we are operating at: (1) with few cores available, enqueue choice adds significant overhead per-task which degrades throughput, and (2) with many cores available, there is little room for improvement between work stealing and single queue. When adding work shedding to work stealing, however, there are some benefits since the shedding mechanism can help balance out queues under high-load conditions when work stealing lacks the extra cycles to help, though the benefit is fairly limited to certain efficiencies as the overheads of flagging can become excessive when spare cycles are rare.

Leveraging hardware. Given these results, we ask two questions motivated by recent advances in hardware: (1) what if the NIC can perform more intelligent distribution than simple hashing? and (2) what impact would handling many cache misses in parallel have? (1) is motivated by recently proposed systems such as the NanoPU [34] which selects queues for incoming packets according to join bounded shortest queue (JBSQ) [43].[6] JBSQ is known to achieve good performance with tail latency improvements up to 10 μs over work stealing, as shown in Figure 3. However, this boost requires new hardware to direct incoming traffic intelligently.

To address (2), we simulated scenarios where the underlying hardware could resolve several cache misses at once (as described in §3.3). Ultimately, this capability means that load-balancing policies may communicate with multiple cores for the price of one (e.g., check 10 cores for the presence of work in work stealing). However, we found that these modifica-

tions provided marginal benefits at best, even assuming that processing the results of parallel checks incurs no overhead.

In general, work stealing was consistently the best performing load-balancing policy when given the same number of cycles as other approaches even as overheads and workload parameters vary. Broadly, work stealing achieves high performance by avoiding per-task overheads and leveraging idle cores to avoid stranding tasks at overloaded cores. Ultimately, absent new hardware, work stealing is the best option for load-balancing approaches among those we evaluated. While work stealing's superiority may seem unsurprising given its widespread use, we believe that we are the first to compare it against other policies and demonstrate its benefits when handling microsecond-scale tasks with realistic load-balancing overheads.

4.2.2 With Dynamic Core Allocations

Next, we consider how load-balancing policies perform when cores can also be reallocated: does reallocating cores change the findings above? When the number of cores allocated to a given application varies over time, it becomes harder to compare approaches (combinations of load-balancing and core-allocation policies). Each combination represents a single point in the tradeoff space between latency and efficiency. If one combination has better latency but worse efficiency than another (i.e., neither is Pareto dominant), which is preferable? Some core-allocation policies are configurable and could be tuned to operate at the same efficiency to compare their latencies. However, not all approaches are tunable (e.g., per-task allocations), so this methodology cannot be used to compare all policies. Thus it is not always possible to say that one policy combination is definitively better than another.

We attempt to pair each load-balancing policy with each other core-allocation policy, but some pairings require modifications or are not reasonable. In Shenango/Caladan, cores park upon failing to find any work to steal. We modify this to work with other load-balancing policies by revoking cores after they spin for the time it would take to check all cores in traditional work stealing, assuming no additional work arrives in the meantime. Per-task allocations maintain the invariant that the number of active cores is equal to the minimum of the number of tasks present and the total number of cores. This is only reasonable with a work-conserving load-balancing policy, so we only evaluate per-task core allocations with the work-stealing load-balancing policy.

With this in mind, we simulated all coherent combinations of load-balancing and core-allocation policies to compare how they explore the available tradeoff space. The results across different average task durations are shown in Figure 5.

Finding 3: *When cores are dynamically reallocated, work stealing performs better than shedding or enqueue choice. This is robust against all factors mentioned in Finding 1.*

Figure 5a shows each combination of load-balancing and core-allocation policies with static-allocation curves for ref-

---

[6]JBSQ(n) queues up to n outstanding tasks at each core (including the task currently being handled) and maintains any surplus in a central queue [43]. We evaluate the case of 3 outstanding tasks, as in NanoPU, though we label this as JBSQ(3) in the terminology of [43] rather than JBSQ(2) as in NanoPU.

(a) 1 μs average task service time.

(b) 10 μs average task service time.

(c) 100 μs average task service time.

Figure 5: Combinations of each core-allocation policy with a load-balancing policy at 50% load with static-allocation curves for each load-balancing policy for reference. Each policy uses the canonical configurations from §4.1. Points for each combination of policies are the color of their load-balancing curve and have the shape type (squares, circles, stars, or triangles) of their core-allocation scheme.
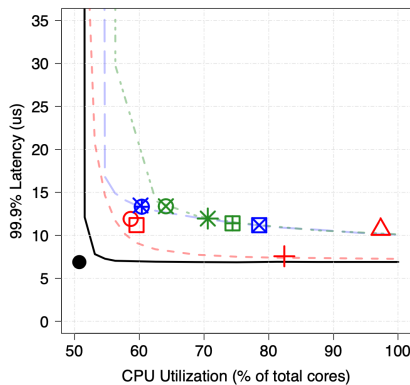
Figure 6: Core-allocation and load-balancing policy combinations for 64 cores and 1 μs tasks. Refer to the legend in Figure 5.

erence. Comparing each core-allocation policy (shape type) across load-balancing policies (colors), we see that work stealing always performs best in terms of proximity to the single queue curve. Note that this graph only shows one choice of parameters for each core-allocation policy, but some can be configured to make different latency vs. efficiency tradeoffs. We generally chose the configuration closest to the bottom-left of the graph, though we will discuss configurability broadly in §4.3.

While adding dynamic core allocations makes the individual performance of each load-balancing policy less clear, overall work stealing still consistently performs better than other load-balancing approaches (absent new hardware).

4.3 Core Allocation

In this section, we compare the performance of different core-allocation policies. Since core-allocation policies are designed to react to changes in load, their performance tends to be tightly coupled with the load-balancing policy employed.

Better load-balancing policies will more effectively use the available cycles, allowing the core-allocation policy to be more conservative in granting cores. Therefore, we evaluate each core-allocation policy across each load-balancing policy and seek to find patterns in the tradeoffs between efficiency and latency that each core-allocation policy makes.

We note that some existing systems use an additional dedicated core (such as Shenango's IOKernel) to perform core allocations [26, 60, 67]. We do not count these cores as we are focusing on policy rather than the implementation of that policy. If we were to include these cores, all efficiency results for these policies would incur an additional 3% CPU utilization for a 32-core system.

We began by asking the question: does reallocating cores yield better performance than sticking with a constant number of cores? One might expect that even with constant average load, being able to react to bursts in load over small time scales would yield significant performance benefits. Surprisingly, we found that the answer to this question is often 'no'.

Finding 4: *For short tasks, none of the core-allocation policies we tried achieved better latency (median or tail) for a given average efficiency than static core allocations (with the same load-balancing policy). However, this becomes possible with longer tasks.*

In Figure 5a, none of the core-allocation policies achieve better tail latency for the same efficiency as a static allocation (the points fall up and to the right of their corresponding static core-allocation curves). As shown in Figures 5b and 5c, when the average task service time is longer (*e.g.,* 10 μs or 100 μs), some policy combinations (points) can achieve better performance than their static-allocation curves. With work stealing and 10 μs service times, delay range, utilization range, and Caladan all beat the static curve for 99.9% tail latency, but

not for the median (omitted for space). This is true for 100 μs service times as well, with per-task also beating the work-stealing curve. As the average task duration increases, the relative importance of the core-allocation overhead decreases and allocating new cores for additional tasks becomes reasonably efficient. The only policy combinations which beat their respective curve include work stealing as the load-balancing policy. Work stealing leverages extra cycles to distribute load while enqueue choice and work shedding are limited in impact since newly added cores will spin idly, unable to handle tasks until a new task arrives or they are flagged by another core.

The only method we have found that can outperform the static curve with tasks as small as 1 μs requires the core-allocation system to be extremely reactive, making core-allocation decisions more frequently than 5 μs and giving the new cores to the application faster than in 5 μs. More frequent allocations are challenging in a real-world implementation because of the overheads of checking state and initiating core reallocations. For example, in Shenango, these actions take roughly 2.1 μs or 3.4 μs with 32 or 64 application cores, respectively [61]. Completing each core reallocation in less than a few microseconds is similarly challenging (§3.3).

Even though core allocations may not provide performance benefits with short tasks, one may employ a core-allocation policy to ensure that the application can adapt to changes in load. Average load in datacenters tends to vary over time [11], so allocating a static number of cores for a constant load would require provisioning for the peak load, wasting CPU cycles over time as load varies. Reacting more slowly to changes in load is also unlikely to perform well; prior work has shown that reactions at 50 ms timescales can cause significant tail latency spikes [60].

Assuming that achieving better performance than the static-allocation curves is unlikely for small tasks, we evaluate the different core-allocation policies in terms of the consistency of their performance and their ability to achieve high CPU efficiency. For some core-allocation policies, the placement relative to a static curve can vary significantly depending on the workload and load, making it difficult for an operator to configure the policy to achieve their goals (e.g., a specific tail latency or CPU efficiency target). By comparing the tradeoffs core-allocation policies make across workloads and loads, we find the policies that exhibit consistent performance.

Finding 5: *Policies that explicitly optimize for an end-to-end user-visible metric (e.g., delay range and utilization range) have more consistent performance, as measured by those metrics, across different configurations.*

For example, Figure 5 illustrates that for Caladan and per-task, the operating point changes with different service times. In contrast, utilization range and delay range specify a range on the x and y axes of the graphs, respectively, that the system should not leave. This generally forces the points to specific
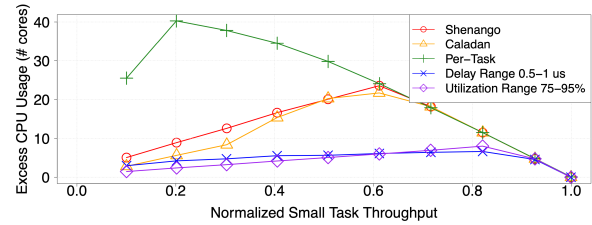
(a) Efficiency measured in excess cores.
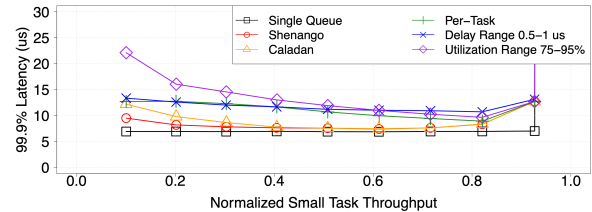
(b) 99.9% latency.

Figure 7: Performance of core-allocation policies paired with work stealing across loads for 64 cores. Efficiency is measured in the excess cores used compared to the single queue simulation.

regions of their static-allocation curves (when the curves cannot be crossed). For example, utilization range points achieve close to 60% CPU utilization across all service times in Figure 5.

Delay range and utilization range also have more predictable performance across different loads. In Figure 7, we illustrate how performance of different core-allocation policies varies with load (when paired with work stealing). We use 64 cores instead of 32 in order to sweep a wider range of loads. Figure 7a shows the efficiency measured as excess cores in comparison to the single queue ideal simulation (i.e., total number of cores used by a given policy minus those used in the ideal case) while Figure 7b shows the tail latency. Caladan, Shenango, and per-task have inconsistent efficiency and tail latency across loads, while delay range and utilization range each keep their respective end-to-end metric relatively constant. Overall, we found that policies such as delay range and utilization range have consistent performance across workloads and configurations, enabling the operator to directly tune the policy's parameters to achieve a specific end-to-end performance objective.

Next, we consider whether each core-allocation policy can be configured to operate near the bend of each static-allocation curve, achieving high CPU efficiency while only minimally compromising in tail latency.

Finding 6: *Yielding cores only when no work is found (when there is no queued work or work stealing fails) makes it challenging to achieve good efficiency with small tasks, especially with many cores.*

The policies that yield cores only when no work is found (Caladan, Shenango, and per-task) cannot always achieve good CPU efficiency, especially with many cores. Here we focus on analyzing each core-allocation policy when paired

with work-stealing, as it performs best. Figure 5a illustrates that per-task achieves poor CPU efficiency with 32 cores, while Figure 6 shows that per-task and Caladan both achieve poor CPU efficiency with 64 cores, using more than 80% of CPU cores for a workload that only requires 50% of cores. In contrast, delay range and utilization range both operate near the bend of the static-allocation curves for 32 and 64 cores. Figure 7 illustrates that utilization range and delay range can save up to 15 cores for similar tail latency across loads.

The efficiency of the Shenango, Caladan, and per-task policies is limited because these policies are slow to yield excess cores. With per-task core allocations, before all cores are allocated the efficiency cannot reach higher than $T/(T+R)$ where T is the average task time and R is the core-allocation overhead, because a core is allocated for every task. Similarly, in policies that yield cores only when work stealing fails (Shenango and Caladan), a significant amount of cycles can be wasted searching through all other cores to never find work or only to find it late in the search. As the number of cores increases, this effect gets worse. Neither Shenango/Caladan nor per-task can be configured to avoid these inefficiencies. Therefore, to achieve high efficiency across workloads and configurations, a core-allocation policy must revoke cores proactively, even when there is or may be some queued work.

We did assess other core-allocation policies such as maintaining a buffer of idle (or work-stealing) cores of a certain size (similar to PerfISO [35]) and enforcing this buffer at every allocation interval. However, this approach tended to be too noisy with short core-allocation intervals and performed significantly worse than other policies.

All together, we found that it is difficult to outperform static core allocations with small tasks, and if the average load is constant and known *a priori*, then statically allocating cores is the best option. However, when load is unknown or changes over time, dynamic allocation policies that proactively revoke cores perform best.

4.4 Policy Takeaways

Overall, our factor analysis found that without new hardware, the best approach is to use work stealing as the load-balancing policy with delay range or utilization range for core allocations, depending on which end-to-end metric is more important to specify and stabilize. Both of these policies are able to operate close to the work-stealing static curve with short tasks or better than the curve with long tasks. Both are robust in the face of service time variability, different service time distributions, load changes, and changes in number of cores. Lastly, both are configurable, allowing the operator to choose whether they prefer CPU efficiency or tail latency (and to what extent). These approaches are intuitive; since core-allocation policies make a tradeoff between CPU efficiency and tail latency, using either parameter effectively as a signal for reallocating cores and controlling where to operate in the space of tradeoffs makes sense.

5 Implementation

We implement our policies in a real system by extending Caladan [75]; our source code is available at `https://github.com/shenango/caladan-policies`. Like its predecessor Shenango [60], Caladan's key components are its application runtime and its dedicated scheduler core, which implements the core-allocation policy. Caladan provides lightweight user-level threading, a high-performance network stack, and load balancing via work stealing. It also enables higher network throughput and its core-allocation mechanisms are more scalable compared to those of Shenango.

We implement both delay range and utilization range atop Caladan. This requires small modifications to both the runtime (50 LOC) and to the scheduler (125 LOC). The Caladan runtime already exposes information about the queueing delay of threads and packets to the scheduler core; we augment this with information about CPU utilization (time spent executing the application vs. in the runtime scheduler) as well. We also add the ability for application cores to yield voluntarily when notified by the scheduler core to do so. When application cores enter the runtime scheduler between tasks, they check if they should yield; for efficiency we do not preempt cores while they are handling tasks. In the scheduler core, we simply add logic for polling the utilization information exposed by applications, and use this or the delay information (depending on the current policy) to decide whether to add or revoke cores. When a core revocation is necessary, the scheduler revokes the core that currently has the least amount of queued work.

Measuring the CPU utilization of application cores over fine timescales is more challenging in practice than in simulation. This is because we do not interrupt running tasks to record CPU usage and only record how CPU time is spent whenever a task starts or finishes. Thus if a task runs for the entirety of a 5 μs core-allocation interval, the scheduler core will observe 0 cycles spent in both the application and the runtime scheduler for that core. The scheduler core handles this by assuming that when application cores report no CPU usage for a core-allocation interval, their utilization is 100%, and it adds a core. In the case when an application has zero allocated CPU cores, CPU utilization is not a useful metric for deciding if an application needs more cores. Thus regardless of the core-allocation policy, the scheduler core always uses the arrival of packets to decide when to grant an application its first core, as in Shenango and Caladan.

6 Evaluation

The goal of our evaluation is to verify that the high-performing policies we identified above can actually yield performance improvements in practice for a real system. Unfortunately, varying the load-balancing policy within a single system would likely involve significant system changes, making a fair comparison difficult (§4). Thus we focus on evaluating the core-allocation policies. We start with a system (Caladan)
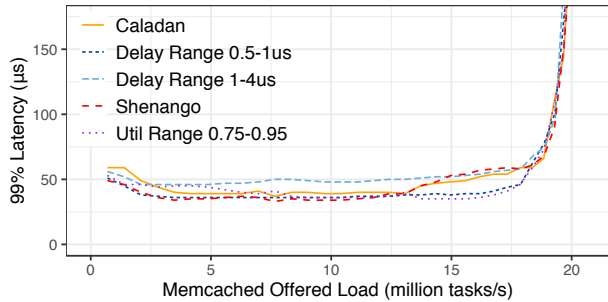
(a) Tail latency for memcached.
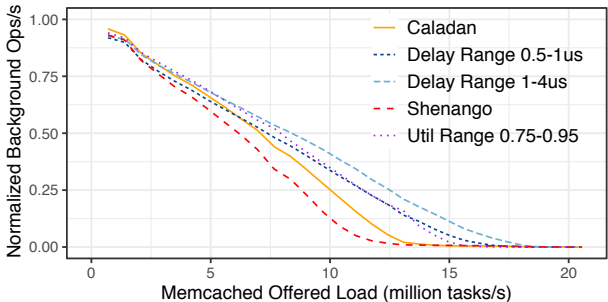
(b) Normalized throughput of the background application.

Figure 8: Performance of two applications under different core-allocation policies, when implemented atop Caladan. The x-axis varies the load of memcached.

that uses the best-performing load-balancing policy (work stealing) and evaluate its performance with different core-allocation policies. We evaluate four policies: Shenango [60], Caladan [26], delay range, and utilization range.

Experimental setup. We conduct experiments using two dual-socket servers with 28-core Intel Xeon Platinum 8176 CPUs operating at 2.1 GHz. Our server machine is equipped with a 40 Gbits/s Mellanox Connect X-5 Bluefield NIC (we do not use the SmartNIC features) and our client machine is equipped with an Intel E810C 100 Gbits/s NIC.[7] We enable hyperthreads and disable TurboBoost and frequency scaling. We use 32 hyperthreads on the second socket (to which our NICs are attached). We use Ubuntu 20.04 with kernel version 5.4.0.

Applications. We evaluate the different policies using *mem-cached* (v1.5.6) [49], a popular key-value store, as our latency-sensitive application. We use *loadgen*, Caladan's open-loop load generator, to generate requests with Poisson arrivals over UDP [75]. Our workload consists of a mixture of read and write requests according to Facebook's USR request distribution [8]; requests have service times of about 1 μs. We run the *swaptions* workload from the PARSEC benchmark suite [13] as a background application and allow it to use all CPU cycles not used by memcached.

---

[7]We run Caladan in "queue steering mode" in which we reconfigure the mappings between NIC queues and cores when core allocations change [61] because our NICs do not support Caladan's default "flow steering mode."

6.1 Policy Comparisons

Our experimental results show that different policies yield different latency vs. CPU efficiency tradeoffs, but that delay range and utilization range generally outperform Shenango and Caladan, confirming the findings of our simulation-based factor analysis. Figure 8a shows the tail latency of mem-cached while Figure 8b shows the throughput achieved by the background application, both as we vary the load offered to memcached (x-axis). We show results for two different configurations of delay range to illustrate the impact of tuning the target range.

In Figure 8, utilization range and delay range (0.5-1 μs) achieve similar tail latency for memcached as Caladan and Shenango, while achieving higher CPU efficiency for the background application. In addition, all of these policies yield similar median latency for memcached (not shown). Shenango is least efficient overall, and these two new policies achieve up to 22% more of the total possible throughput for the background application (7 hyperthreads worth) than Shenango. Compared to Caladan, these policies achieve up to 13% more throughput for the background application (4 hyperthreads worth). This is because with Shenango and Caladan's policies, memcached spends much more time in the runtime scheduler, primarily work stealing (up to 26% and 21% of its time, respectively). In contrast, with the other policies, CPU time in the scheduler is much lower. For example, with utilization range, memcached spends less than 14% of its time in the scheduler at all except the lowest loads. By proactively revoking unused cores rather than waiting for work stealing to fail to find tasks to handle, delay range and utilization range can achieve higher CPU efficiency without degrading the performance for memcached.

Both the delay range and utilization range policies take as input a target range, and these ranges can be adjusted to make different tradeoffs between tail latency and CPU efficiency. As an example, Figure 8 shows two different ranges for delay range. Delay range 1-4 μs achieves about 2 hyperthreads worth of additional throughput for the batch application compared to delay range 0.5-1 μs, at the cost of 10-15 μs of tail latency.

7 Related Work

Load-balancing policies. Load-balancing policies have been studied extensively, both theoretically and in the context of real systems. Several systems have adopted the ideal policy of maintaining a single shared queue [36, 62], though they experience throughput bottlenecks as a result. Others take the opposite approach and perform no load balancing in software, leaving it to the NIC [12, 64] or storage device [42] to randomly distribute work across cores; these approaches suffer from load imbalances.

Work stealing was originally proposed as a way of efficiently scheduling multithreaded computations across multi-

ple cores [14,39,71]. Variants of work stealing have been studied thoroughly [54,57] and adopted in task-parallel platforms such as OpenMP [59], IntelTBB [68], Cilk [47], Habanero [9, 16], X10 [17], Java Fork/Join [45], and the Go runtime [76]. More recently, work stealing has been adopted in datacenter systems as a way to provide low tail latency [26,44,60,66,81]. Similarly, past work has analyzed the power-of-two choices load-balancing policy [55] as well as variants of it, such as those that consider known service times [31] or general service time distributions [15]. Arachne [67], SKQ [81], and many other systems [29, 33, 63, 82] leverage power-of-two or the more general power-of-k choices for load balancing. Others have studied work shedding approaches [73] and compared them to other policies [22,77]. Finally, several recent proposals implement more advanced load-balancing policies such as JBSQ [43] in NIC hardware [18,34,70,74].

Our findings are consistent with past comparisons of load-balancing policies. For example, we confirm that "work-first" load-balancing policies such as work stealing have better performance [21, 22, 27]. However, our analysis differs in two key ways. First, we are not aware of any prior work that compares load-balancing policies in the presence of realistic load-balancing overheads; prior work either assumes no overhead or analyzes a single system and its policy and overheads. Second, prior work evaluates metrics such as delay, throughput, and communication rate, but does not consider CPU efficiency. In contrast, we compare the tradeoffs that different policies make in terms of latency and efficiency, in the presence of load-balancing overheads.

Core-allocation policies. Existing systems adopt a variety of different policies for deciding when to reallocate cores, either across different applications or between cores available for applications and those designated for network processing or a file system. These approaches make decisions based on task arrivals [40], queueing delay [12,20,26,52,53,60,67], CPU utilization [12,20,35,41,67], or failure to find work [4,7,21, 27,40,76]. None of these systems compare different policies in the presence of the same overheads, so it is not possible to determine from these works which policies provide the best combination of latency and efficiency. Some past work points out that work-stealing cores can waste considerable CPU cycles, and proposes policies for yielding cores to mitigate this [4,7,21]. However, these policies target throughput and fairness for longer tasks (e.g., hundreds of microseconds or more); in contrast, our analysis focuses on which policies provide the best efficiency and latency for microsecond-scale tasks, and thus yields different conclusions.

Implementing policies. The systems Syrup [37] and ghOSt [32] enable users to control scheduling policies in the kernel scheduler, network stack, and network card from code written in userspace. These systems are complementary to our work; they make it easier to express scheduling policies but do not specify which policies users should implement.

8 Conclusion

Numerous systems have been designed to support latency-sensitive datacenter applications while dynamically allocating cores to react to changes in load. However, these systems often come with a significant efficiency penalty with short tasks. In this paper, we systematically evaluated the effects of different policy choices on efficiency and latency to determine which load-balancing and core-allocating schemes achieve the best performance when considering realistic overheads. Work stealing is the definitive best policy option in today's hardware while the core-allocation space is more complex. We designed and implemented two core-allocation policies which provide consistent and configurable performance on the Pareto frontier when paired with work stealing and demonstrated how they can improve efficiency without sacrificing latency.

9 Acknowledgments

References

[1] Dpdk. https://www.dpdk.org/.

[2] redis. https://redis.io/.

[3] Rss. https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-receive-side-scaling.

[4] K. Agrawal, C. E. Leiserson, Y. He, and W. J. Hsu. Adaptive work-stealing with parallelism feedback. *ACM Transactions on Computer Systems (TOCS)*, 26(3):1–32, 2008.

[5] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center tcp (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 63–74, 2010.

[6] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 263–274, 2018.

[7] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of computing systems*, 34(2):115–144, 2001.

[8] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference*

on Measurement and Modeling of Computer Systems, SIGMETRICS '12, page 53–64, New York, NY, USA, 2012. Association for Computing Machinery.

[9] R. Barik, Z. Budimlic, V. Cavè, S. Chatterjee, Y. Guo, D. Peixotto, R. Raman, J. Shirako, S. Taşırlar, Y. Yan, et al. The habanero multicore software research project. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 735–736, 2009.

[10] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan. Attack of the killer microseconds. *Communications of the ACM*, 60(4):48–54, 2017.

[11] L. A. Barroso and U. Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 4(1):1–108, 2009.

[12] A. Belay, G. Prekas, M. Primorac, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. The ix operating system: Combining low latency, high throughput, and efficiency in a protected dataplane. *ACM Transactions on Computer Systems (TOCS)*, 34(4):1–39, 2016.

[13] C. Bienia. *Benchmarking modern multiprocessors*. Princeton University, 2011.

[14] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[15] M. Bramson, Y. Lu, and B. Prabhakar. Randomized load balancing with general service time distributions. *ACM SIGMETRICS performance evaluation review*, 38(1):275–286, 2010.

[16] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-java: the new adventures of old x10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, pages 51–61, 2011.

[17] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *ACM SIGPLAN Notices*, 40(10):519–538, 2005.

[18] A. Daglis, M. Sutherland, and B. Falsafi. Rpcvalet: Ni-driven tail-aware balancing of μs-scale rpcs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 35–48, 2019.

[19] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[20] H. M. Demoulin, J. Fried, I. Pedisich, M. Kogias, B. T. Loo, L. T. X. Phan, and I. Zhang. When idling is ideal: Optimizing tail-latency for heavy-tailed datacenter workloads with perséphone. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*, pages 621–637, 2021.

[21] X. Ding, K. Wang, P. B. Gibbons, and X. Zhang. Bws: balanced work stealing for time-sharing multicores. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 365–378, 2012.

[22] D. L. Eager, E. D. Lazowska, and J. Zahorjan. A comparison of receiver-initiated and sender-initiated adaptive load sharing. *Performance evaluation*, 6(1):53–68, 1986.

[23] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *2011 38th Annual international symposium on computer architecture (ISCA)*, pages 365–376. IEEE, 2011.

[24] Z. Fang, S. Mehta, P.-C. Yew, A. Zhai, J. Greensky, G. Beeraka, and B. Zang. Measuring microarchitectural details of multi-and many-core memory systems through microbenchmarking. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(4):1–26, 2015.

[25] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 363–376, 2017.

[26] J. Fried, Z. Ruan, A. Ousterhout, and A. Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297, 2020.

[27] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 212–223, 1998.

[28] M. R. Garey and D. S. Johnson. *Computers and intractability*, volume 174. freeman San Francisco, 1979.

[29] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 649–667, 2017.

[30] T. Hellemans, T. Bodas, and B. Van Houdt. Performance analysis of workload dependent load balancing policies. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 3(2):1–35, 2019.

[31] T. Hellemans and B. Van Houdt. On the power-of-d-choices with least loaded server selection. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2(2):1–22, 2018.

[32] J. T. Humphries, N. Natu, A. Chaugule, O. Weisse, B. Rhoden, J. Don, L. Rizzo, O. Rombakh, P. Turner, and C. Kozyrakis. ghost: Fast & flexible user-space delegation of linux scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*, pages 588–604, 2021.

[33] J. Hwang, M. Vuppalapati, S. Peter, and R. Agarwal. Rearchitecting linux storage stack for μs latency and high throughput. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 113–128. USENIX Association, July 2021.

[34] S. Ibanez, A. Mallery, S. Arslan, T. Jepsen, M. Shahbaz, C. Kim, and N. McKeown. The nanopu: A nanosecond network stack for datacenters. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 239–256, 2021.

[35] C. Iorgulescu, R. Azimi, Y. Kwon, S. Elnikety, M. Syamala, V. Narasayya, H. Herodotou, P. Tomita, A. Chen, J. Zhang, et al. Perfiso: Performance isolation for commercial latency-sensitive services. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 519–532, 2018.

[36] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 345–360, 2019.

[37] K. Kaffes, J. T. Humphries, D. Mazières, and C. Kozyrakis. Syrup: User-defined scheduling across the stack. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*, pages 605–620, 2021.

[38] A. Kalia, M. Kaminsky, and D. Andersen. Datacenter rpcs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 1–16, 2019.

[39] R. M. Karp and Y. Zhang. Randomized parallel algorithms for backtrack search and branch-and-bound computation. *Journal of the ACM (JACM)*, 40(3):765–789, 1993.

[40] M. Karsten and S. Barghi. User-level threading: Have your cake and eat it too. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 4(1):1–30, 2020.

[41] A. Kaufmann, T. Stamler, S. Peter, N. K. Sharma, A. Krishnamurthy, and T. Anderson. Tas: Tcp acceleration as an os service. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.

[42] A. Klimovic, H. Litz, and C. Kozyrakis. Reflex: Remote flash ≈ local flash. *ACM SIGARCH Computer Architecture News*, 45(1):345–359, 2017.

[43] M. Kogias, G. Prekas, A. Ghosn, J. Fietz, and E. Bugnion. R2P2: Making RPCs first-class datacenter citizens. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 863–880, 2019.

[44] C. Kulkarni, S. Moore, M. Naqvi, T. Zhang, R. Ricci, and R. Stutsman. Splinter: Bare-metal extensions for multi-tenant low-latency storage. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 627–643, 2018.

[45] D. Lea. A java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, pages 36–43, 2000.

[46] C. Lee and J. Ousterhout. Granular computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 149–154, 2019.

[47] C. E. Leiserson. The cilk++ concurrency platform. *The Journal of Supercomputing*, 51(3):244–257, 2010.

[48] D. Lemire. Code used on daniel lemire's blog. https://github.com/lemire/Code-used-on-Daniel-Lemire-s-blog/tree/master/2019/01/01.

[49] J. Leverich and C. Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–14, 2014.

[50] J. Li, K. Agrawal, S. Elnikety, Y. He, I.-T. A. Lee, C. Lu, and K. S. McKinley. Work stealing for interactive services to meet target latency. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–13, 2016.

[51] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, 2014.

[52] J. Liu, A. Rebello, Y. Dai, C. Ye, S. Kannan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Scale and performance in a filesystem semi-microkernel. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*, pages 819–835, 2021.

[53] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. Gribble, et al. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 399–413, 2019.

[54] M. Mitzenmacher. Analyses of load stealing models based on differential equations. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 212–221, 1998.

[55] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[56] M. Nandagopal, K. Gokulnath, and V. R. Uthariaraj. Sender initiated decentralized dynamic load balancing for multi cluster computational grid environment. In *Proceedings of the 1st Amrita ACM-W Celebration on Women in Computing in India*, pages 1–4. 2010.

[57] D. Neill and A. Wierman. On the benefits of work stealing in shared-memory multiprocessors. *Department of Computer Science, Carnegie Mellon University, Tech. Rep*, 2009.

[58] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association.

[59] OpenMP. Openmp application programming interface. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf, 2018.

[60] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, 2019.

[61] A. E. Ousterhout. *Achieving high CPU efficiency and low tail latency in datacenters*. PhD thesis, Massachusetts Institute of Technology, 2019.

[62] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, et al. The ramcloud storage system. *ACM Transactions on Computer Systems (TOCS)*, 33(3):1–55, 2015.

[63] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, pages 69–84, 2013.

[64] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16, 2014.

[65] A. Pourhabibi, M. Sutherland, A. Daglis, and B. Falsafi. Cerebros: Evading the rpc tax in datacenters. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 407–420, 2021.

[66] G. Prekas, M. Kogias, and E. Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, pages 325–341, 2017.

[67] H. Qin, Q. Li, J. Speiser, P. Kraft, and J. Ousterhout. Arachne: core-aware thread management. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 145–160, 2018.

[68] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. 2007.

[69] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay. AIFM: High-performance, application-integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 315–332, 2020.

[70] A. Rucker, M. Shahbaz, T. Swamy, and K. Olukotun. Elastic RSS: Co-scheduling packets and cores using programmable NICs. In *Proceedings of the 3rd Asia-Pacific Workshop on Networking 2019*, pages 71–77, 2019.

[71] L. Rudolph, M. Slivkin-Allalouf, and E. Upfal. A simple load balancing scheme for task allocation in parallel machines. In *Proceedings of the third annual ACM symposium on Parallel algorithms and architectures*, pages 237–245, 1991.

[72] H. Schweizer, M. Besta, and T. Hoefler. Evaluating the cost of atomic operations on modern architectures. https://arxiv.org/pdf/2010.09852.pdf.

[73] N. G. Shivaratri, P. Krueger, and M. Singhal. Load distributing for locally distributed systems. *Computer*, 25(12):33–44, 1992.

[74] M. Sutherland, S. Gupta, B. Falsafi, V. Marathe, D. Pnevmatikatos, and A. Daglis. The nebula rpc-optimized

architecture. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 199–212. IEEE, 2020.

[75] The Caladan Authors. Caladan's open-source release. https://github.com/shenango/caladan.

[76] The Go Community. The go programming language. https://golang.org.

[77] B. Van Houdt. Randomized work stealing versus sharing in large-scale systems with non-exponential job sizes. *IEEE/ACM Transactions on Networking*, 27(5):2137–2149, 2019.

[78] R. V. Van Nieuwpoort, T. Kielmann, and H. E. Bal. Efficient load balancing for wide-area divide-and-conquer applications. In *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 34–43, 2001.

[79] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–17, 2015.

[80] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. Cpi2: Cpu performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 379–391, 2013.

[81] S. Zhao, H. Gu, and A. J. Mashtizadeh. Skq: Event scheduling for optimizing tail latency in a traditional os kernel. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 759–772, 2021.

[82] H. Zhu, K. Kaffes, Z. Chen, Z. Liu, C. Kozyrakis, I. Stoica, and X. Jin. RackSched: A Microsecond-Scale scheduler for Rack-Scale computers. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1225–1240. USENIX Association, Nov. 2020.

A   Appendix

A.1   Proof of NP-Hardness for Optimal Core Allocations

In this appendix, we prove that finding the optimal tail latency for a given CPU usage bound or vice versa is NP-hard, assuming a finite number of cores and non-constant service times. We show this using a reduction from the multiprocessor scheduling decision problem.

Multiprocessor Scheduling Problem [28]

Input: A non-zero number of cores c, a set of tasks T where each task t has a positive integer service time (or length) $l(t)$, and an overall deadline D for completing all tasks.

Question: Is there a schedule of the tasks T over the c cores that meets the overall deadline D? Such a schedule assigns a start time to each task t such that there are never more than c tasks being handled simultaneously and for each task, its start time plus $l(t)$ is at most D.

The Multiprocessor Scheduling Problem is NP-complete, assuming that all tasks do not have the same service time; with constant service times, this problem is trivial [28].

Optimal Core-Allocation Problem

Input: A non-zero number of cores c where each core can be either *on* or *off*, and transitioning from off to on requires a start-up time of S; a set of tasks T where each task t has an arrival time $a(t)$ and a positive integer service time $l(t)$; the total "wasted" CPU time W, or time spent by cores while they are starting up or on but not handling a task; a tail latency percentile $P < 1$ (e.g., 99.9[th] percentile); and a tail latency target L.

Question: Is there a schedule for the c cores and the tasks T such that tasks are only scheduled on cores that are *on*, the wasted CPU time is at most W, and the latency at percentile P is at most L? A schedule for the cores assigns periods of *on* and *off* time to each, noting that it takes S time to transition from *off* to *on*. A schedule for the tasks assigns a start time to each task t such that the start time for t is at least $a(t)$ and the number of tasks being handled simultaneously never exceeds the number of cores that are *on*. Finally, for P percent of the tasks, their start time plus $l(t)$ is at most L.

We can reduce the multiprocessor scheduling problem to the optimal core allocation problem as follows. The number of cores in the core allocation problem matches that in the multiprocessor scheduling problem and we set $L = D$. We construct the set of tasks for the core allocation problem by replicating the tasks and their service times from the multiprocessor scheduling problem and setting them to all arrive at the beginning (i.e., $a(t) = 0$ for all $t \in T$). In addition, we add additional dummy tasks with $l(t) > D$ so that the tasks in the multiprocessor scheduling problem constitute P percent of the total tasks in the core allocation problem; because the dummy tasks cannot possibly meet the latency bound, the problem is only solvable by having all non-dummy tasks meet the latency bound. Finally we set the start-up time S to be zero and the

(a) 30% load.

(b) 70% load.
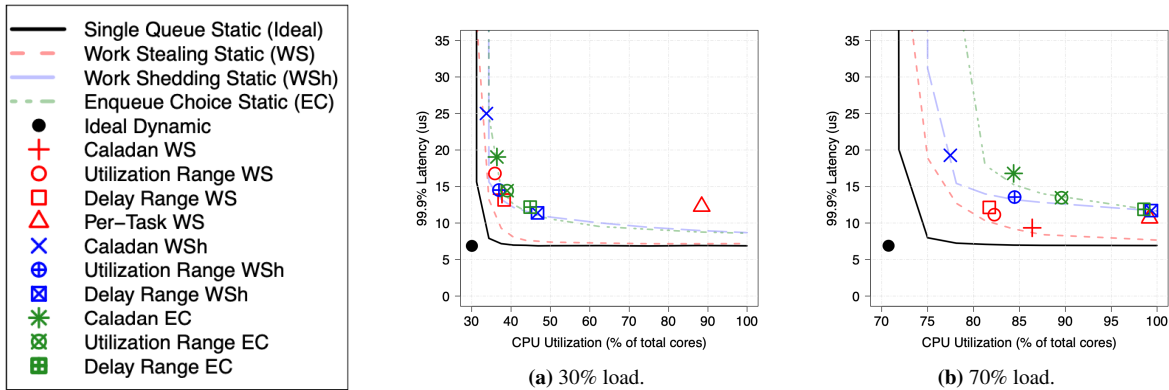
Figure 9: Performance of core-allocation and load-balancing policies from Figure 5 at additional loads.

wasted CPU time bound W to be high enough to be irrelevant (e.g., $W \geq c \cdot D$). We leave it as a simple exercise to show that there exists a polynomial time algorithm for such an instance of the optimal core allocation problem if and only if there is a polynomial time algorithm for the corresponding instance of the multiprocessor scheduling problem. In addition, the optimal core allocation problem is clearly in NP; thus it is NP-complete.

Because the optimal core allocation decision problem is NP-complete, the optimization problem of finding the optimal tail latency for a given efficiency bound or vice versa is NP-hard. This proof assumes that the service time distribution $l(t)$ is not constant; the optimization problem with constant service times may also be NP-hard but this cannot be shown using the proof above.

A.2 Extended Factor Analysis

In this appendix we include additional data omitted for space in the factor analysis.

A.2.1 Additional Loads for Static Curves

In Figure 5, we compared the performance of different load-balancing policies across different average service times to demonstrate that beating static allocations is more difficult with short tasks. The graphs look at both efficiency and latency simultaneously by keeping load constant. In Figure 9, we vary the offered load to 30% and 70%. To see a complete view of efficiency and latency (without static load-balancing curves for reference) across load, see Figure 7.

A.2.2 Additional Service Time Distributions

We compared the load-balancing policies across different service time distributions. Specifically, we created static allocation performance curves for each load-balancing policy for both constant service times of 1 μs and a bimodal distribution with 500 ns service times for 90% of requests and 5.5 μs for the remaining 10% (average service time of 1 μs). In Figure 10, we see that across these different service time distributions, work stealing consistently outperforms the other

load-balancing policies. Since load-balancing choices are less significant to end-to-end performance when service times are constant (Figure 10a), work stealing provides smaller benefits.

(a) Constant service times (1 μs).

(b) Bimodal service time distribution.
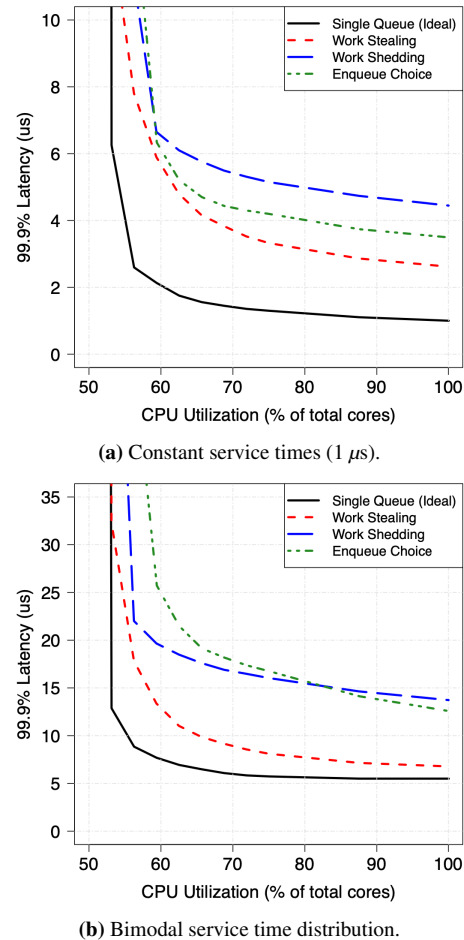
Figure 10: Performance of load-balancing policies with static core allocations for different service time distributions.

A Case for Task Sampling based Learning for Cluster Job Scheduling

Akshay Jajoo*
akshay.jajoo@nokia-bell-labs.com

Y. Charlie Hu
ychu@purdue.edu

Xiaojun Lin
linx@purdue.edu

Nan Deng
dengnan@google.com

Abstract

The ability to accurately estimate job runtime properties allows a scheduler to effectively schedule jobs. State-of-the-art online cluster job schedulers use history-based learning, which uses past job execution information to estimate the runtime properties of newly arrived jobs. However, with fast-paced development in cluster technology (in both hardware and software) and changing user inputs, job runtime properties can change over time, which lead to inaccurate predictions.

In this paper, we explore the potential and limitation of real-time learning of job runtime properties, by proactively sampling and scheduling a small fraction of the tasks of each job. Such a task-sampling-based approach exploits the similarity among runtime properties of the tasks of the same job and is inherently immune to changing job behavior. Our analytical and experimental analysis of 3 production traces with different skew and job distribution shows that learning in space can be substantially more accurate. Our simulation and testbed evaluation on Azure of the two learning approaches anchored in a generic job scheduler using 3 production cluster job traces shows that despite its online overhead, learning in space reduces the average Job Completion Time (JCT) by 1.28×, 1.56×, and 1.32× compared to the prior-art history-based predictor. Finally, we show how sampling-based learning can be extended to schedule DAG jobs and achieve similar speedups over the prior-art history-based predictor.

1 Introduction

In big-data compute clusters, jobs arrive online and compete to share the cluster resources. In order to best utilize the cluster and to ensure that jobs also meet their service level objectives, efficient scheduling is essential. However, as jobs arrive online, their runtime characteristics are not known a priori. Due to this lack of information, it is challenging for the cluster scheduler to determine the right job execution order that optimizes scheduling metrics such as maximal resource utilization or application service level objectives.

An effective way to tackle the challenges of cluster scheduling is to learn the runtime characteristics of pending jobs, which allows the scheduler to exploit offline scheduling algorithms that are known to be optimal, *e.g.,* Shortest Job First (SJF) for minimizing the average completion time. Indeed, there has been a large amount of work [27, 36, 43, 44, 47, 49,

---

52, 55] on learning job runtime characteristics to facilitate cluster job scheduling.

In essence, all of the previous learning algorithms learn job runtime characteristics from observing historical executions of the same jobs, which execute the same code but process different sets of data, or of similar jobs, which have matching features such as the same application name, the same job name, or the same user who submitted the job.

The effectiveness of the above *history-based* learning schemes critically rely on two conditions to hold true: (1) The jobs are recurring; (2) The performance of the same or similar jobs will remain consistent over time.

In practice, however, the two conditions often do not hold true. First, many previous work have acknowledged that not all jobs are recurrent. For example, in the traces used in Corral [43] and Jockey [30], only 40% of the jobs are recurrent, and Morpheus [44] shows that only 60% of the jobs are recurrent. Second, even the authors of history-based prediction schemes such as 3Sigma [47] and Morpheus [44] strongly argued why runtime properties of jobs, even with the same input, will not remain consistent and will keep evolving. The primary reason is due to updates in cluster hardware, application software, and user scripts to execute the cluster jobs. Third, our own analysis of three production cluster traces (§4) have also shown that historical job runtime characteristics have considerable variations.

In this paper, we explore an alternative approach to learning runtime properties of distributed jobs online to facilitate cluster job scheduling. The approach is motivated by the following key observations about distributed jobs running on shared clusters: (1) a job typically has a *spatial dimension*, *i.e.,* it typically consists of many tasks; and (2) the tasks (in the same phase) of a job typically execute the same code and process different chunks of similarly sized data [9, 16]. These observations suggest that if the scheduler first schedules a few sampled tasks of a job, known as pilot tasks, to run till finish, it can use the observed runtime properties of those tasks to accurately estimate those of the whole job. Effectively, such a *task-sampling-based* approach learns job properties in the spatial dimension. We denote the new learning scheme as SLEARN, for "learning in space".

Intuitively, by using the execution of pilot tasks to predict the properties of other tasks, SLEARN avoids the primary drawback of history-based learning techniques, *i.e.,* relying on jobs to be recurring and job properties to remain stationary over time. However, learning in space introduces two new challenges: (1) its estimation accuracy can be affected

---

by the variations of task runtime properties, *i.e.,* task skew; (2) delaying scheduling the remaining tasks of a job till the completion of sampled tasks may potentially hurt the job's completion time.

In this paper, we perform a comprehensive comparative study of history-based learning (learning in time) and sampling-based learning (learning in space), to systematically answer the following questions: *(1) Can learning in space be more accurate than learning in time? (2) If so, can delaying scheduling the remaining tasks of a job till the completion of sampled tasks be more than compensated by the improved accuracy and result in improved job performance,* e.g., *completion time?*

We answer the first question via quantitative analysis, and trace and experimental analysis based on three production job traces, including two public cluster traces from Google released in 2011 and 2019 [8, 11] and a private trace from 2Sigma [1]. We answer the second question by designing a generic scheduler that schedules jobs based on job runtime estimates to optimize a given performance metric, *e.g.,* average job completion time (JCT), and then plug into the scheduler different prediction schemes, in particular, learning in time and learning in space, to compare their effectiveness.

We summarize the major findings and contributions of this paper as follows:

- Based on literature survey and analysis using three production cluster traces, we show that history is not a stable and accurate predictor for runtime characteristics of distributed jobs.

- We propose SLEARN, a novel learning approach that uses sampling in the spatial dimension of jobs to learn job runtime properties online. We also provide solutions to practical issues such as dealing with thin jobs (jobs with a few tasks only) and work conservation.

- Via quantitative, trace and experimental analysis, we demonstrate that SLEARN can predict job runtime properties with much higher accuracy than history-based schemes. For the 2Sigma, Google 2011, and Google 2019 cluster traces, the median prediction error are 18.98%, 13.68%, and 51.84% for SLEARN but 36.57%, 21.39%, and 71.56% for the state-of-the-art history-based 3Sigma, respectively.

- We show that learning job runtime properties by sampling job tasks, although delays scheduling the remaining tasks of a job, can be more than compensated by the improved accuracy, and as a result reduces the average JCT. In particular, our extensive simulations and testbed experiments using a prototype on a 150-node cluster in Microsoft Azure show that compared to the prior-art history-based predictor, SLEARN reduces the average JCT by 1.28×, 1.56×, and 1.32× for the extracted 2Sigma, Google 2011 and Google 2019 traces, respectively.

- We show how the sampling-based learning can be extended to schedule DAG jobs. Using a DAG trace generated from the Google 2019 trace, we show a hybrid sampling-based and history-based scheme reduces the average JCT by 1.25× over a pure history-based scheme.

2 Background and Related Work

In this section, we provide a brief background on the cluster scheduling problem, review existing learning-based schedulers, and discuss their weaknesses.

2.1 Cluster Scheduling Problem

In both public and private clouds, clusters are typically shared among multiple users to execute diverse jobs. Such jobs typically arrive online and compete for shared resources. In order to best utilize the cluster and to ensure that jobs also meet their service level objectives (SLOs), efficient job scheduling is essential. Since jobs arrive online, their runtime characteristics are not known a priori. This lack of information makes it challenging for the scheduler to determine the right order for running the jobs that maximizes resource utilization and/or meets application SLOs. Additionally, jobs have different SLOs. For some meeting deadlines is important while for others faster completion or minimizing the use of networks is more important. Such a diverse set of objectives pose further challenges to effective job scheduling [19, 30, 31, 43, 44, 55, 56].

2.2 Job Model

We consider big-data compute clusters running data-parallel frameworks such as Hadoop [4], Hive [6], Dryad [37], Scope [22], and Spark [7] that run simple MapReduce jobs [28] or more complex DAG-structured jobs, where each job processes a large amount of data. Each job consists of one or multiple stages, such as map or reduce, and each stage partitions the data into manageable chunks and runs many parallel tasks, each for processing one data chunk.

2.3 Existing Learning-based Schedulers

An effective way to tackle the challenges of cluster scheduling is to learn runtime characteristics of pending jobs. As such cluster schedulers using various learning methods have been proposed [19, 21, 25, 36, 43–45, 47, 49, 50, 52]. In essence, all previous learning schemes are *history-based*, *i.e.,* they learn job characteristics by observations made from the past job executions.[1] In particular, existing learning approaches can be broadly categorized into the following groups, as summarized in Table 1.

Learning offline models. Corral's prediction model is designed with the primary assumptions that most jobs are

---

[1]Some recent work use the characteristics of completed mini-batches as a proxy for the remaining mini-batches, to improve the scheduling of ML jobs [54]. However, such jobs are different in that the mini-batches in general experience significantly less (task-level) variations than what we studied in this paper.

Table 1: Summary of selected previous work that use history-based learning techniques.

Name	Property estimated	Estimation technique	Learning frequency
Corral [43]	Job runtime	Offline model (not updated)	On arrival
DCOSR [36]	Memory elasticity profile	Offline model (not updated)	Scheduler dependent
Jockey [30]	Job runtime	Offline simulator	Periodic
3Sigma [47]	Job runtime history dist.	Offline model	On arrival

recurring in nature, and the latency of each stage of a multi-stage job is proportional to the amount of data processed by it, which do not always hold true [43].

DCOSR [36] predicts the memory usage for data parallel compute jobs using an offline model built from a fixed number of profile runs that are specific to the framework and depend on the framework's properties. Any software update in the existing frameworks, addition of new framework or hardware update will require an update in profile.

For analytics jobs that perform the same computation periodically on different sets of data, Tetris [32] takes measurements from past executions of a job to estimate the requirements for the current execution.

Learning offline models with periodic updates. Jockey [30] periodically characterizes job progress at runtime, which along with a job's current resource allocation is used by an offline simulator to estimate the job's completion time and update the job's resource allocation. Jockey relies on job recurrences and cannot work with new jobs.

Learning from similar jobs. Instead of using execution history from the exact same jobs, JVuPredict [51] matches jobs on the basis of some common features such as application name, job name, the user who owns the job, and the resource requested by the job. 3Sigma [47] extends JVuPredict [51] by introducing a new idea on prediction: instead of using point metrics to predict runtimes, it uses full distributions of relevant runtime histories. However, since it is impractical to maintain precise distributions for each feature value, it resorts to approximating distributions, which compromises the benefits of having full distributions.

2.4 Learning from History: Assumptions and Reality

Predicting job runtime characteristics from history information relies on the following two conditions to hold, which we argue may not be applicable to modern day clusters.

Condition 1: The jobs are recurring. Many previous works have acknowledged that not all jobs are recurrent. For example, the traces used in Corral [43] and Jockey [30] show that only 40% of the jobs are recurrent and Morpheus [44] shows that 60% of the jobs are recurrent.

Condition 2: The performance of the same or similar jobs will remain consistent over time. Previous works [30, 43, 44, 47] that exploited history-based prediction have considered jobs in one of the following two categories. (1) *Recurring jobs*: A job is re-scheduled to run on newly arriving data; (2) *Similar jobs:* A job has not been seen before but has some attributes in common with some jobs executed in the past [47, 51]. Many of the history-based approaches only predict for recurring jobs [30, 43, 44], while some others [25, 45, 47, 51] work for both categories of jobs.

However, even the authors of history-based prediction schemes such as 3Sigma [47] and Morpheus [44] strongly argued why runtime properties of jobs, even with the same input, will keep evolving. The primary reason is that updates in cluster hardware, application software, and user scripts to execute the cluster jobs affect the job runtime characteristics. They found that in a large Microsoft production cluster, within a one-month period, applications corresponding to more than 50% of the recurring jobs were updated. The source code changed by at least 10% for applications corresponding to 15-20% of the jobs. Additionally, over a one-year period, the proportion of two different types of machines in the cluster changed from 80/20 to 55/45. For a same production Spark job, there is a 40% difference between the running time observed on the two types of machines [44].

For these reasons, although the state-of-the-art history-based system 3Sigma [47] uses sophisticated prediction techniques, the predicted running time for more than 23% of the jobs have at least 100% error, and for many the prediction is off by an order of magnitude.

3 SLEARN – Learning in Space

In this paper, we explore an alternative approach to learning job runtime properties online in order to facilitate cluster job scheduling. The approach is motivated by the following key observations about distributed jobs running in shared clusters: (1) a distributed job has a spatial dimension, *i.e.,* it typically consists of many tasks; (2) all the tasks in the same phase of a job typically execute the same code with the same settings [9, 12, 16], and differ in that they process different chunks of similarly sized data. Hence, it is likely that their runtime behavior will be statistically similar.

The above observations suggest that if the scheduler first schedules a few sampled tasks of a job to run till finish, it can use the observed runtime properties of those tasks to accurately estimate those of the whole job. In a modular design, such an online learning scheme can be decoupled from the cluster scheduler. In particular, upon a job arrival, the predictor first schedules sampled tasks of the job, called *pilot tasks*, till their completion, to learn the job runtime properties. The learned job properties are then fed into the cluster job scheduler, which can employ different scheduling polices to meet respective SLOs. Effectively, the new scheme learns job properties in the spatial dimension, *i.e.,learning in*

Table 2: Comparison of learning in time and learning in space of job runtime properties.

	Applicability	Adaptiveness	Accuracy	Runtime overhead
Time	Recurring jobs	No/Yes	Depends	No
Space	New/Recurring jobs	Yes	Depends	Yes

space. We denote the new learning scheme as SLEARN.

Table 2 summarizes the pros and cons of the two learning approaches along four dimensions: (1) **Applicability:** As discussed in §2.3, most history-based predictors cannot be used for the jobs of a new category or for categories for which the jobs are rarely executed. In contrast, learning in space has no such limitation; it can be applied to any new job. (2) **Adaptiveness to change:** Further, history-based predictors assume job runtime properties persist over time, which often does not hold, as discussed in §2.4. (3) **Accuracy:** The accuracy of the two approaches are directly affected by how they learn, *i.e.,* in space versus in time. The accuracy of history-based approaches is affected by how stable the job runtime properties persist over time, while that of sampling-based approach is affected by the variation of the task runtime properties, *i.e.,* the extent of task skew. (4) **Runtime overhead:** The history-based approach has an inherent advantage of having very low to zero runtime overhead. It performs offline analysis of historical data to generate a prediction model. In contrast, sampling-based predictors do not have offline cost, but need to first run a few pilot tasks till completion before scheduling the remaining tasks. This may potentially delay the execution of non-sampled tasks.

The above qualitative comparison of the two learning approaches raises the following two questions: *(1) Can learning in space be more accurate than learning in time? (2) If so, can delaying scheduling the remaining tasks of a job till the completion of sampled tasks be more than compensated by the improved accuracy, so that the overall job performance,* e.g., *completion time, is improved?* We answer the first question via analytical, trace and experimental analysis in §4 and the second question via a case study of cluster job scheduling using the two types of predictors in §5.

4 Accuracy Analysis

In this section, we perform an in-depth study of the prediction accuracy of the two learning approaches: *learning in time* (history-based learning) and *learning in space* (task-sampling-based learning). Both approaches can potentially be used to learn different job properties for different optimization objectives. In this paper, we focus on job completion time because it is an important metric that has been intensively studied in recent work [23, 24, 29, 33, 35, 36, 43, 47].

4.1 Analytical Comparison

We first present a theoretical analysis of the prediction accuracies of the two approaches. We caution that here we use a highly-stylized model (e.g., two jobs and normal task-length

distributions), which does not capture the possible complexity in real clusters, such as heavy parallelism across servers and highly-skewed task-length distributions. Nonetheless, it reveals important insights that help us understand in which regimes history-based schemes or sampling-based schemes will perform better. Consider a simple case of two jobs j_1 and j_2, where each job has n tasks. The size of each task of j_1 is known. Without loss of generality, let us assume that the task size of j_1 is 1. Thus, the total size of j_1 is n. The size of a task of j_2 is however unknown. Let x denote the average task size of j_2, and this its total size is nx. Clearly, if we knew x precisely, then we should have scheduled j_1 first if $x > 1$ and j_2 first if $x \leq 1$. However, suppose that we only know the following: (1) (Prior distribution:) x follows a normal distribution with mean μ and variance σ_o^2; (2) Given x, the size of a random task of the job follows a normal distribution with mean x and variance σ_1^2. Intuitively, σ_o^2 captures the variation of mean task-lengths *across* many *i.i.d.* copies of job j_2, *i.e.,* job-wise variation, while σ_1^2 captures the variation of task-lengths *within* a single run of job j_2, *i.e.,* task-wise variation. We note that the parameters σ_o^2 and σ_1^2 are *not* used by the predictors below.

Now, consider two options for estimating the mean task-length x:1) A history-based approach (§4.1.1) and (2) a sampling-based approach where we sample m tasks from j_2 (§4.1.2).

4.1.1 History-based Schemes

Since no samples of job j_2 are used, the best predictor for its mean task length is μ. In other words, the scheduling decision will be based on μ only. The difference between the true mean task length, x, and μ is simply captured by the job-wise variance σ_o^2.

4.1.2 Sampling-based Schemes

Suppose that we sample m tasks from j_2. Collect the sampled task lengths into a vector:

$$\vec{y} = (y_1, y_2, ..., y_m).$$

Then, based on our probabilistic model, we have

$$P(y_i|x) = \frac{1}{\sqrt{2\pi}\sigma_1}e^{-\frac{(y_i-x)^2}{2\sigma_1^2}}, \quad P(\vec{y}|x) = \prod_{i=1}^{m}\frac{1}{\sqrt{2\pi}\sigma_1}e^{-\frac{(y_i-x)^2}{2\sigma_1^2}}$$

We are interested in an estimator of x given $\vec{y}$. We have

$$P(x|\vec{y}) = \frac{P(\vec{y}|x)\cdot P(x)}{P(\vec{y})} = \frac{P(\vec{y}|x)\cdot P(x)}{\int_x P(\vec{y}|x)\cdot P(x)dx}$$

$$= \frac{1}{\sqrt{2\pi}}\left[\frac{m}{\sigma_1^2}+\frac{1}{\sigma_o^2}\right]^{\frac{1}{2}}\cdot e^{-\left(\frac{m}{2\sigma_1^2}+\frac{1}{2\sigma_o^2}\right)\left(x-\frac{\sum_{i=1}^{m}\frac{1}{\sigma_1^2}y_i+\frac{1}{\sigma_o^2}\mu}{\frac{m}{\sigma_1^2}+\frac{1}{\sigma_o^2}}\right)},$$

where the last step follows from standard results on the posterior distribution with Gaussian priors (see, *e.g.,* [18]). In other words, conditioned on $\vec{y}$, x also follows a normal distribution with mean $=\frac{\sum_{i=1}^{m}\frac{1}{\sigma_1^2}y_i+\frac{1}{\sigma_o^2}\mu}{\frac{m}{\sigma_1^2}+\frac{1}{\sigma_o^2}}$ and variance $=\frac{1}{\frac{m}{\sigma_1^2}+\frac{1}{\sigma_o^2}}$.

Table 3: Summary of trace properties.

Trace	Arrival time	Resource requested	Resource usage	Indiv. task duration
2Sigma	Yes	Yes	No	Yes
Google 2011	Yes	Yes	Yes	Yes
Google 2019	Yes	Yes	Yes	Yes

Note that this represents the estimator quality using the information of both job-wise variations and task-wise variations. If the estimator is not informed of the job-wise variations, we can take $\sigma_o^2 \to +\infty$, and the conditional distribution of x given $\vec{y}$ becomes normal with mean $\frac{1}{m}\sum_{i=1}^{m} y_i$ and variance $\frac{\sigma_1^2}{m}$.

From here we can draw the following conclusions. First, whether history-based schemes or sampling-based schemes have better prediction accuracy for an unknown job depends on the relationship between job-wise variations σ_o^2 and the task-wise variation σ_1^2. If the job-wise variation is large but the task-wise variation is small, *i.e.,* $\sigma_o^2 >> \frac{\sigma_1^2}{m}$, then sampling-based schemes will have better prediction accuracy. Conversely, if the job-wise variation is small but the task-wise variation is large, *i.e.,* $\sigma_o^2 << \frac{\sigma_1^2}{m}$, then history-based schemes will have better prediction accuracy. Second, while the accuracy of history-based schemes is fixed at σ_o^2, the accuracy of sampling-based schemes improves as m increases. Thus, when we can afford the overhead of more samples, the sampling-based schemes become favorable. Our results from experimental data below will further confirm these intuitions.

4.2 Trace-based Variability Analysis

Our theoretical analysis in §4.1 provides insights on how the prediction accuracies of the two approaches depend on the variation of job run times across time and space. To understand how such variations fare against each other in practice, we next measure the actual variations in three production cluster traces. Table 3 summarizes the information available in the traces that are used in our analysis.

Traces. Our first trace is provided by 2Sigma [1]. The cluster uses an internal proprietary job scheduler running on top of a Mesos cluster manager [2]. This trace was collected over a period of 7 months, from January to July 2016, and from 441 machines and contains approximately 0.4 million jobs [17].

We also include two publicly available traces from Google released in May 2011 and May 2019 [8, 11], collected from 1 and 8 Borg [53] cells over periods of 29 and 31 days, respectively. The machines in the clusters are highly heterogeneous, belonging to at least three different platforms that use different micro-architectures and/or memory technologies [20]. Further, according to [9], the machines in the same platform can have substantially different clock rates, memory speed, and core counts. Since the original Google 2019 trace has data from 8 different cells located in 8 different locations,

and given that we already have two other traces from the US, we chose the batch tier of Cluster G in the Google 2019 trace, which is located in Singapore [12], as our third trace to diversify our trace collection.

We calculate the variations in task runtimes for each job across time and across space as follows.

Variation across time. To measure the variation in mean task runtime for a job across the history, we follow the following prediction mechanism defined in 3Sigma [47] to find similar jobs.

As discussed in §2.3, 3Sigma [47] uses multiple features to identify a job and predicts its runtime using the feature that gives the least prediction error in the past. We include all six features used in 3Sigma: application name, job name, user name (the owner of the job), job submission time (day and hour), and resources requested (cpu and memory) by the job.

For each feature, we define the set of similar jobs as all the jobs executed in the history window (defined below) that had the same feature value. Next, we calculate the average task runtime of each job in the set. Then, we calculate the *Coefficient of Variation* (CoV) of the average task runtimes across all the jobs in the set. We repeat the above process for all the features. We then compare the CoV values thus calculated and pick the minimum CoV. Effectively, the above procedure selects the least possible variation across history.

Varying the history length in prediction across time. 3Sigma used the entire history for prediction. Intuitively, the length of the history affects the trade-off between the number of similar jobs and the staleness of the history information. For this reason, we optimized 3Sigma by finding and using the history length that gives the least variation. Specifically, we define the length of history based on a window size w, *i.e.,* the number of past consecutive days. In our analysis below, we vary w among 3, 7, and 14 for the three traces.

Variation across space. To measure the extent of variation across space, we look at the CoV (CoV $= \frac{\sigma}{\mu}$) in the task runtimes within a job. As shown in §4.1, the variance in the task runtime predicted from sampling is $\frac{\sigma_1^2}{m}$, where σ_1^2 is the variance in the runtimes across all the tasks within the job and m is the number of tasks sampled. Thus, we first estimate σ_1^2 from all tasks within the job. We then report the CoV of our task runtime prediction after sampling m tasks as $\frac{\sigma_1/\sqrt{m}}{\mu}$. Our complete scheduler design in §5.1 uses an adaptive sampling algorithm which mostly uses 3% for the three traces. Thus, for measuring the extent of variation across space here, we assume a 3% sampling ratio and plot $\frac{\sigma_1}{(\sqrt{0.03 \times numberOfTasksInJob}) \times \mu}$.

Variability comparison. For consistency, all analysis results here are for the same, shortest trace period that can be used for sliding-window-history based analysis, *e.g.,* the last 15 days under the 14-day window for the 29-day Google 2011 trace. (The analysis then varies the length of the sliding window in history-based learning.)
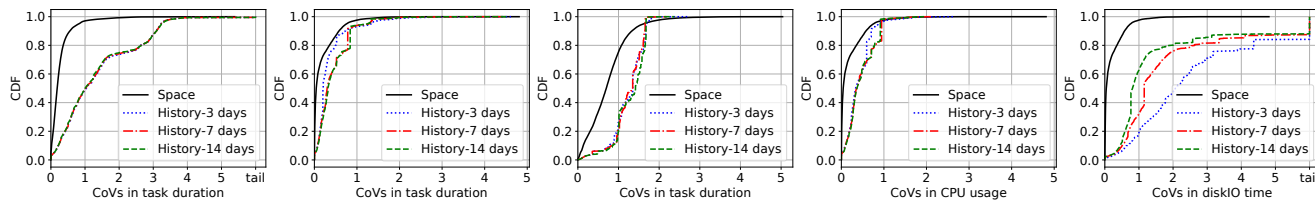
(a) Task runtime – 2Sigma (b) Task runtime – Google 11 (c) Task runtime – Google 19 (d) CPU usage – Google 11 (e) Disk IO time – Google 11

Figure 1: CDF of CoV of runtime properties across space and across time with varying history windows, using the 2Sigma, Google 2011 and Google 2019 traces. Single-task jobs are excluded from the analysis across space.

Table 4: CoV in task runtime across time and across space for the the 2Sigma, Google 2011, and Google 2019 traces.

Trace	CoV over Time		CoV over Space	
	P50	P90	P50	P90
2Sigma	1.00	3.10	0.18	0.55
Google 2011	0.20	0.73	0.04	0.58
Google 2019	1.35	1.67	0.70	1.33

Fig. 1(a)–Fig. 1(c) show the CDFs of CoVs in task duration measured across space and across history for multiple history window sizes for the three traces. We see that in general using a shorter sliding window reduces the prediction error of 3Sigma, and the CoVs across tasks are moderately lower than the CoVs across history for the Google 2011 trace but significantly lower for 2Sigma and Google 2019 traces. For example, for the 2Sigma trace, the CoV across history is higher than the CoV across tasks for 85.40% of the jobs (not seen in Fig. 1(a) as jobs are ordered differently in different CDFs) and for more than 30% of the jobs, the CoV across history is at least $12.10\times$ higher than the CoV across tasks.

Table 4 summarizes the results, where the CoVs across time correspond to the best history window size, *i.e.*, 3 days for both Google traces and 14 days for the 2Sigma trace. As shown in the table, the P50 (P90) CoV across history are 1.00 (3.10) for the 2Sigma trace, 0.20 (0.73) for the Google 2011 trace, and 1.35 (1.67) for the Google 2019 trace. In contrast, the P50 (P90) CoV value across the task duration of the same set of jobs is much lower, 0.18 (0.55) for the 2Sigma trace, 0.04 (0.58) for the Google 2011 trace, and 0.70 (1.33) for the Google 2019 trace.

Fig. 1(d) and Fig. 1(e) further show the CDF of CoVs for CPU usage and Disk IO time for the Google 2011 trace (such resource usage is not available in the 2Sigma trace). The figures show that the variation in the values of these properties when sampled across space is also considerably lower compared to the variation observed over time.

4.3 Experimental Prediction Error Analysis

Recall from our analysis in §4.1 that lower task-wise variation than job-wise variation (§4.2) will translate into better prediction accuracy of sampling-based schemes over history-based schemes. While our analysis in §4.1 assumes normal distribution, we believe that a similar conclusion will hold

in more general settings. To validate this, we next implement a sampling-based predictor SLEARN, and experimentally compare it against a state-of-the-art history-based predictor 3Sigma [47] in estimating the job runtimes directly on production job traces.

Workload characteristics. Since the three production traces described in §4.2 are too large, as in 3Sigma [47], we extracted smaller traces for experiments using the procedure described below.

Since the history-based predictor 3Sigma needs a history trace, we followed the same process as in [47] to extract the training trace for 3Sigma and the execution trace for all predictors, in three steps. (1) We divided each original trace in chronological order in two halves. (2) We compressed 2Sigma jobs to 150 tasks or fewer, by applying a compression ratio of original cluster size/150. Since the Google traces do not have many wide jobs yet the original clusters are very wide, with 12.5K machines, we dropped jobs with more than 150 tasks [2]. (3) We next selected the execution trace following the process below from the second half; these became 2STrace, GTrace11 and GTrace19, respectively. (4) We then selected jobs from the first half of each original trace that are feature-clustered with those jobs in the execution trace to form the "history" trace for 3Sigma.

We extracted the execution trace from each of the above-mentioned second halves by randomly selecting 1250 jobs with equal probability. Then, for each extracted trace, we adjust the arrival time of the jobs so that the average cluster load matches that in the original trace [8, 11, 17]. Table 5 summarizes the workload per window of the extracted traces, where a window is defined as a 1000-second interval sliding by 100 seconds at a time, and the load per window is the total runtime of all the jobs arrived in that window, normalized by the total number of CPUs in the cluster times the window length, *i.e.*, 1000s. We see that for all three traces, the average system load is close to 1, though the load fluctuates over time, which is preserved by the random uniform job extraction.

Prediction mechanisms and experimental setups. We implement the 3Sigma predictor following its description

---

[2]This is to avoid potential bias towards SLEARN. A job with more than 150 tasks will have to be scheduled in more than one phase, which will be in favor of SLEARN by diminishing the sampling overhead.

Table 5: Statistics for system load per 1000s sliding window.

Trace	Average	P50	P90
2STrace	1.05	0.13	2.47
GTrace11	1.01	0.29	1.49
GTrace19	1.04	0.09	0.91

in [47]. After learning the job runtime distribution (§4.2), it uses a utility function of the estimated job runtime associated with every job to derive its estimated runtime from the distribution, by integrating the utility function over the entire runtime distribution. Since our goal is to minimize the average JCT, we used a utility function that is inversely proportional to the square of runtime. We kept all the default settings we learned from the authors of 3Sigma [47].

As in §4.2, SLEARN samples $max(1, 0.03 \cdot S)$ tasks per job, where S is the number of tasks in the job. We only show the results for wide jobs (with 3 or more tasks) as in the complete SLEARN design (§5.1.1), only wide jobs go through the sampling phase.

Results. Fig. 2 shows the CDF of percentage error in the predicted job runtimes for the three traces. We see that SLEARN has much better prediction accuracy than 3Sigma. For 2STrace, GTrace11, and GTrace19, the P50 prediction error are 18.30%, 9.15%, 21.39% for SLEARN but 36.57%, 21.39%, 71.56% for 3Sigma, respectively, and the P90 prediction error are 58.66%, 49.95%, 92.25% for SLEARN but 475.78%, 294.52%, 1927.51% for 3Sigma, respectively.

5 Integrating Sampling-based Learning with Job Scheduling: A Case Study

In this section, we answer the second key question about the sampling-based learning: Can delaying scheduling the remaining tasks till completing the sampled tasks be compensated by the improved prediction accuracy? We answer it through extensive simulation and testbed experiments.

Our approach is to design a generic scheduler, denoted as GS, that schedules jobs based on job runtime estimates to optimize a given performance metric, average job completion time (JCT). We then plug into GS different prediction schemes to compare their end-to-end performance.

5.1 Scheduler and Predictor Design

5.1.1 Generic Scheduler GS

GS replaces the scheduling component of a cluster manager like YARN [5]. The key scheduling objective of GS is to minimize the average JCT. Additionally, GS aims to avoid starvation.

The scheduling task in GS is divided into two phases, (1) job runtime estimation, and (2) efficient and starvation-free scheduling of jobs whose runtimes have been estimated. We focus here on the scheduling mechanism and discuss the different job runtime estimators in the following sections.

Inter-job scheduling. Shortest job first (SJF) is known to be optimal in minimizing the average JCT when job execution depends on a single resource. Previous work has shown that scheduling distributed jobs even with prior knowledge is NP-hard (*e.g.,* [24]), and an effective online heuristic is to order the distributed jobs based on each job's total size [23, 39–41]. In GS we use a similar heuristic; the jobs are ordered based on their total estimated runtime, *i.e., mean task runtime × number of tasks.*

Starvation avoidance. SJF is known to cause starvation to long jobs. Hence, in GS we adopt a well-known multi-level priority queue structure to avoid job starvation [23, 26, 38, 46, 48]. Once GS receives the runtime estimates of a job, it assigns the job to a priority queue based on its runtime. Within a queue, we use FIFO to schedule jobs. Across the queues, we use weighted sharing of resources, where a priority queue receives a resource share according to its priority.

In particular, GS uses N queues, Q_0 to Q_{N-1}, with each queue having a lower queue threshold Q_q^{lo} and a higher threshold Q_q^{hi} for job runtimes. We set $Q_0^{lo} = 0$, $Q_{N-1}^{hi} = \infty$, $Q_{q+1}^{lo} = Q_q^{hi}$. A queue with a lower index has a higher priority. GS uses exponentially growing queue thresholds, *i.e.,* $Q_{q+1}^{hi} = E \cdot Q_q^{hi}$. To avoid any bias, we use the multiple priority queue structure with the same configuration when comparing different job runtime estimators.

Basic scheduling operation. GS keeps track of resources being used by each priority queue. It offers the next available resource to a queue such that the weighted sharing of resources among the queues for starvation avoidance is maintained. Resources offered to a queue are always offered to the job at the head of the queue.

5.1.2 SLEARN

To seamlessly integrate SLEARN with GS, we need to use one of the priority queues for scheduling sampled tasks. We denote it as the sampling queue.

Fast sampling. One design challenge is how to determine the priority for the sampling queue w.r.t. the other priority queues. On one hand, sampled tasks should be given high priority so that the job runtime estimation can finish quickly. On the other hand, the jobs whose runtimes have already been estimated should not be further delayed by learning new jobs. To balance the two factors, we use the second highest priority in GS as the sampling queue.

Handling thin jobs. Recall that in SLEARN, when a new job arrives, SLEARN only schedules its pilot tasks, and delays other tasks until the pilot tasks finish and the job runtime is estimated. Such a design choice can inadvertently lead to higher JCTs for thin jobs, *e.g.,* a two-task job would experience serialization of its two tasks. To avoid JCT degradations for thin jobs, we place a job directly in the highest priority queue if its width is under a threshold thinLimit.
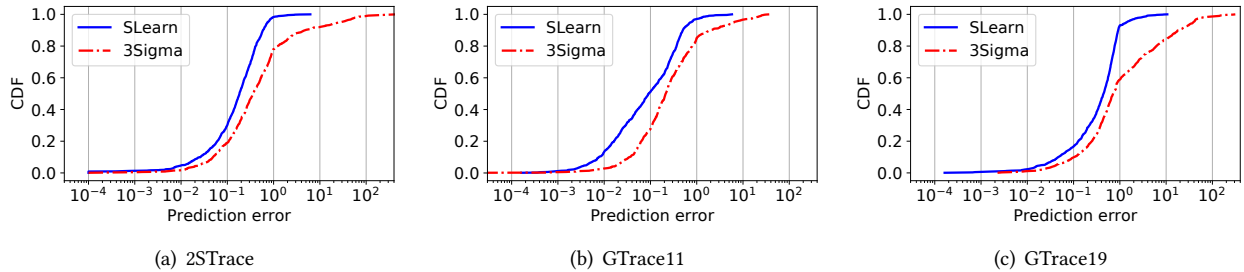
Figure 2: Job runtime prediction accuracy.

Basic operations. Upon the arrival of a new job, the cluster manager asynchronously communicates the job's information to GS, which relays the information to SLEARN. If the number of tasks in the job is under thinLimit, SLEARN assigns it to the highest priority queue; otherwise, the job is assigned to the sampling queue, where a subset of its tasks (*pilot tasks*) will be scheduled to run. Once a job's runtime is estimated from sampling, it is placed in the priority queue corresponding to its runtime estimate where the rest of its tasks will be scheduled.

How many and which pilot tasks to schedule? When a new job arrives, SLEARN first needs to determine the number of pilot tasks. Sampling more tasks can give higher estimation accuracy, but also consumes more resources early on, which can potentially delay other jobs, if the job turns out to be a long job and should have been scheduled to run later under SJF. Further, we found the best sampling ratio appears to vary across difference traces. To balance the trade-off, we use an adaptive algorithm to dynamically determine the sampling ratio, as shown in Figure 3. The basic idea of the algorithm is to suggest a sampling ratio that has resulted in the lowest job completion time normalized by the job runtime based on the recent past. To achieve this, for every value in a defined range of possible sampling ratios (between 1% and 5%), it maintains a running score (*srScoreMap*), which is the average normalized JCT of T recently finished jobs that used the corresponding sampling ratio. In practice we found a T value of 100 works reasonably well. During system start-up, it tries sampling ratios of 2%, 3%, and 4% for the first $3T$ jobs (Line 2–7). It further tries sampling ratios of 1% and 5% if going down from 3% to 2% or going up from 3% to 4% reduces the normalized JCT. Afterwards, for each new job, it uses the sampling ratio that has the lowest running score. Finally, upon each job completion, the score map is updated (Line 16–24).

Once the sampling ratio is chosen, SLEARN selects pilot tasks for a job randomly.

How to estimate from sampled tasks? Several methods such as bootstrapping, statistical mean or median can be used to predict job properties from sampled tasks. In GS, we use empirical mean to predict the mean task runtime.

Work conservation. When the system load is low, some

```
1:  procedure GETCURRENTSAMPLINGPERCENTAGE(Job j)
2:      if j in First T jobs then
3:          return 3
4:      else if j in Second T jobs then
5:          return 2
6:      else if j in Third T jobs then
7:          return 4
8:      minScore = getMinValue(srScoreMap)
9:      if minScore.SR == 2 then
10:         if 1.1*minScore.value < srScoreMap[3].value then
11:             return 1
12:     if minScore.SR == 4 then
13:         if srScoreMap[3].value > 1.1*minScore.value then
14:             return 5
15:     return minScore.SR
16: procedure UPDATESCOREONJOBCOMPLETION(Job j)
17:     sr = j.sr                          ▷ Get j's sampling ratio.
18:     normalizedJCT = j.jct          ▷ Get j's normalized JCT.
19:     UpdateScoresMap(sr, normalizedJCT)
20: procedure UPDATESCOREMAPS(sr, normalizedJCT)
21:     if Len(jobWiseSrScoresMap[sr])>T then
22:         Drop first element of jobWiseSrScoresMap[sr]
23:     jobWiseSrScoresMap[sr].append(normalizedJCT)
24:     srScoreMap[sr].value = mean(jobWiseSrScoresMap[sr])
```

Figure 3: Adaptive sampling algorithm in SLEARN.

machines may be idle while the non-sampling tasks are waiting for the sampling tasks to finish. In such cases, SLEARN schedules non-sampling tasks of jobs to run on otherwise idle machines. In work conservation, the jobs are scheduled in the FIFO order of their arrival.

5.1.3 Baseline Predictors and Policies

We compare SLEARN's effectiveness against four different baseline predictors and two policies: **(1) 3Sigma:** as discussed in §4.3. **(2) 3SigmaTL:** same as 3Sigma but handles thin jobs in the same way as SLEARN; they are directly placed in the highest priority queue. This is to isolate the effect of thin job handling. **(3) POINT-EST:** same as 3Sigma, with the only difference being that, instead of integrating a utility function over the entire runtime history, it predicts a point estimate (median in our case) from the history. **(4)** LAS**:** The Least Attained Service [48] policy approximates SJF online

without explicitly learning job sizes, and is most recently implemented in the Kairos [29] scheduler. LAS uses multiple priority queues and the priority is inversely proportional to the service attained so far, *i.e.,* the total execution time so far. We use the sum of all the task execution time to be consistent with all the other schemes. **(5)** FIFO: The FIFO policy in YARN simply prioritizes jobs in the order of their arrival. Since FIFO is a starvation free policy, there is no need for multiple priority queues. **(6)** ORACLE: ORACLE is an ideal predictor that always predicts with 100% accuracy.

5.2 Experimental Results

We evaluated SLEARN's performance against the six baseline schemes discussed above by plugging them in GS and execute the 3 traces (2STrace, GTrace11, and GTrace19) using large scale simulations and on a 150-node testbed cluster in Azure (§5.2.6).

5.2.1 Experimental Setup

Cluster setup. We implemented GS, SLEARN and baseline estimators with 11 KLOC of Java and python2. We used an open source java patch for Gridmix [15] and open source java implementation of NumericHistogram [13] for Hadoop. We used some parts from DSS, an open source job scheduling simulator [10], in simulation experiments.

We implemented a proxy scheduler wrapper that plugs into the resource manager of YARN [5] and conducted real cluster experiments on a 150-node cluster in MS Azure [14].

Following the methodology in recent work on cluster job scheduling [25,47,51], we implemented a synthetic generator based on the Gridmix implementation to replay jobs that follow the arrival time and task runtime from the input trace. The Yarn master runs on a standard DS15 v2 server with 20-core 2.4 GHz Intel Xeon E5-2673 v3 (Haswell) processor and 140GB memory, and the slaves run on D2v2 with the same processor with 2-core and 7GB memory.

Parameters. The default parameters for priority queues in GS in the experiments are: starting queue threshold (Q_0^{hi}) is 10^6 ms, exponential threshold growth factor (E) is 10, number of queues (N) is set to 10, and the weights for time sharing assigned to individual priority queues decrease exponentially by a factor of 10. Previous work (*e.g.,* [23]) and our own evaluation have shown that the scheduling results are fairly insensitive to these configuration parameters. We omit their sensitivity study here due to page limit. SLEARN chooses the number of pilot tasks for wide jobs using the adaptive algorithm described in §5.1.2 and the threshold for thin jobs is set to 3. We evaluate the effectiveness of adaptive sampling in §5.2.2 and the sensitivity to thinLimit in §5.2.8.

Performance metrics. We measure three performance metrics in the evaluation: JCT speedup, defined as the ratio of a JCT under a baseline scheme over under SLEARN, the job runtime estimation accuracy, and job waiting time.

Table 6: Performance improvement of SLEARN over 3Sigma under adaptive sampling and fixed-ratio sampling.

	Fraction of tasks chosen as pilot tasks						
	1%	2%	3%	4%	5%	10%	Adap.
	2STrace						
P50 pred. error (%)	19.4	19.0	19.0	18.7	18.4	16.9	19.0
Avg. JCT speedup (×)	1.24	1.23	1.27	1.26	1.27	1.28	1.28
P50 speedup (×)	0.93	0.92	0.93	0.92	0.93	0.91	0.92
	GTrace11						
P50 pred. error (%)	14.4	14.0	13.6	13.1	12.7	9.09	13.7
Avg. JCT speedup (×)	1.52	1.55	1.54	1.56	1.58	1.51	1.56
P50 speedup (×)	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	GTrace19						
P50 pred. error (%)	55.7	53.8	47.1	46.5	42.1	36.1	51.8
Avg. JCT speedup (×)	1.31	1.31	1.31	1.32	1.28	1.24	1.32
P50 speedup (×)	1.07	1.07	1.05	1.05	1.01	1.00	1.07

Workload. We used the same training data for history-based estimators and the test traces (2STrace, GTrace11 and GTrace19) as described in §4.3.

5.2.2 Effectiveness of Adaptive Sampling

In this experiment, we evaluate the effectiveness of our adaptive algorithm for task sampling. Fig. 4 shows how the sampling ratio selected by the adaptive algorithm for each job varies between 1% and 5% over the duration of the three traces. We further compare average JCT speedup and P50 speedup under the adaptive algorithm with those under a fixed sampling ratio, ranging between 1% and 10%. Table 6 shows that the adaptive sampling algorithm leads to the best speedups for 2STrace and GTrace19 and is about only 1% worse than the best for GTrace11. Interestingly, we observe that no single sampling ratio works the best for all traces. Nonetheless, the adaptive algorithm always chooses one that is the best or closest to the best in terms of JCT speedup. More importantly, we see that the adaptive algorithm does not always use the sampling ratio with the best prediction accuracy, which shows that it effectively balances the tradeoff between prediction accuracy and sampling overhead.

5.2.3 Prediction Accuracy

SLEARN achieves more accurate estimation of job runtime over 3Sigma – the details were already discussed in §4.3.

5.2.4 Average JCT Improvement

We now compare the JCT speedups achieved using SLEARN over using the five baseline schemes defined in §5.1.3.

Fig. 5(a) shows the results for 2STrace. We make the following observations. (1) Compared to ORACLE, SLEARN achieves an average and P50 speedups of 0.79× and 0.73×, respectively. This is because SLEARN has some estimation error; it places 10.91% of wide jobs in the wrong queues, 3.54% in lower queues and 7.37% in higher queues. (2) SLEARN improves the average JCT over 3Sigma by 1.28×. This significant improvement of SLEARN comes from much higher prediction accuracy compared to 3Sigma (Fig. 2). (3) The

(a) 2STrace

(b) GTrace11

(c) GTrace19
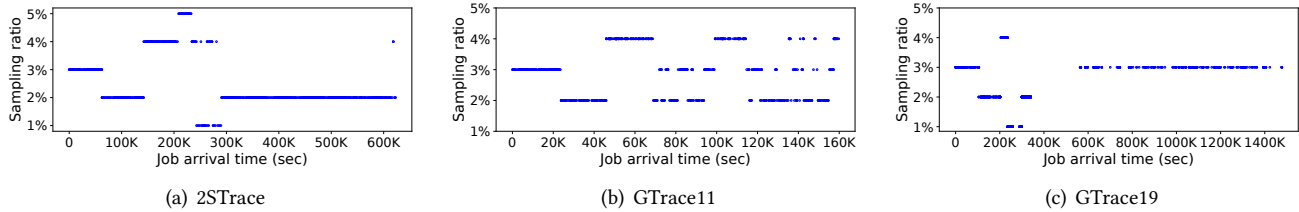
Figure 4: Sampling ratios selected by the adaptive sampling algorithm. The duration of initial $3T$ jobs appear varying due to uneven arrival times.

(a) 2STrace

(b) GTrace11
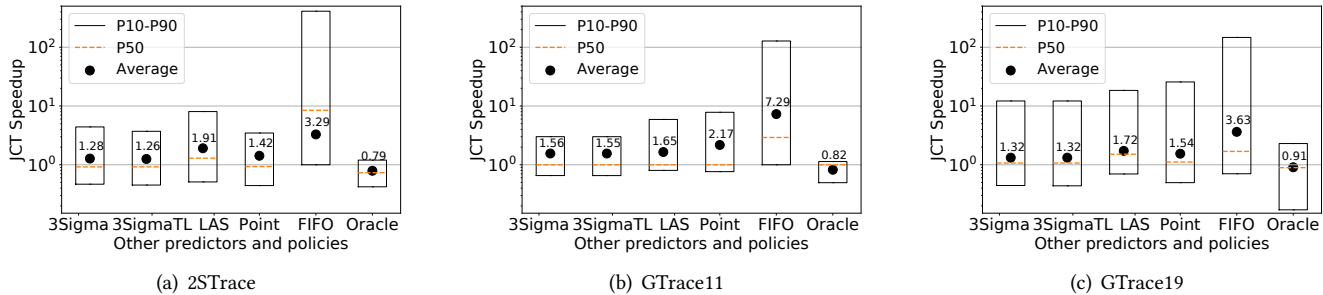
(c) GTrace19

Figure 5: JCT speedup using SLEARN as compared to other baseline schemes for the three traces.

Table 7: Percentage of the wide jobs that had correct queue assignment.

Prediction Technique	SLEARN	3Sigma
2STrace	89.09%	73.84%
GTrace11	86.45%	76.20%
GTrace19	73.96%	58.07%

improvement of SLEARN over 3SigmaTL, 1.26×, is similar to that over 3Sigma, confirming thin job handling only played a small role in the performance difference of the two schemes. To illustrate SLEARN's high prediction accuracy, we show in Table 7 the fraction of wide jobs that were placed in correct queues by SLEARN and 3Sigma. We observe that SLEARN consistently assigns more wide jobs to correct queues than 3Sigma for all three traces. (4) Compared to POINT-EST, SLEARN improves the average JCT by 1.42×. Again, this is because SLEARN estimates runtimes with higher accuracy. (5) Compared to LAS, SLEARN achieves an average JCT speedup of 1.91× and P50 speedup of 1.29×. This is because LAS pays a heavy penalty in identifying the correct queues of jobs by moving them across the queues incrementally. (6) Lastly, compared with FIFO, SLEARN achieves an average JCT speedup of 3.29× and P50 speedup of 8.45×.

Fig. 5(b) shows the results for GTrace11. Scheduling under SLEARN again outperforms all other schemes. In particular, using SLEARN improves the average JCT by 1.56× compared to using 3Sigma, 1.55× compared to using 3SigmaTL, 2.17× compared to using Point-Est, and 1.65× compared to using the LAS policy. Fig. 5(c) shows that scheduling under SLEARN outperforms all other schemes for GTrace19 too. In particular, using SLEARN improves the average JCT by 1.32×, 1.32×, 1.54×, and 1.72× compared to using 3Sigma, 3SigmaTL, POINT-EST and the LAS policy, respectively.

In summary, our results above show that SLEARN's higher estimation accuracy outweighs its runtime overhead from sampling, and as a result achieves much lower average job completion time than history-based predictors and the LAS policy for the three production workloads.

5.2.5 Impact of Sampling on Job Waiting Time

To gain insight into why sampling pilot tasks first under SLEARN does not hurt the overall average JCT, we next compare the *normalized waiting time* of jobs, calculated as the average waiting time of its tasks under the respective scheme, divided by the mean task length of the job.

Fig. 6 shows the CDF of the normalized job waiting time under SLEARN and 3Sigma. We see that the CDF curves can be divided into three segments. (1) The first segment, where both SLearn and 3Sigma have normalized waiting time (NWT) less than 0.04, covers 36.58% of the jobs, and 35.57% of the jobs are common. The jobs have almost identical NWT, much lower than 1 under both schemes. This happens because during low system load periods, *e.g.,* lower than 1, the scheduler will schedule all the tasks to run under both scheme; under SLEARN it schedules non-sampled tasks of jobs to run before their sampled tasks complete due to work conservation. (2) The second segment, where both schemes have NWT between 0.04 and 1.90, covers 30.51% of the jobs, and 20.38% of the jobs are common. Out of these 20.38%, 29.81% have lower NWT under SLEARN and 70.19% have lower NWT under 3Sigma. This happens because when the system load is moderate, the jobs experience longer waiting time under SLEARN than under 3Sigma because of sampling delay. (3) The third segment, where both schemes have NWT above 1.90, cover 32.91% of the jobs, and 24.68% of jobs are common. Out of these 24.68%, 83.08% have lower waiting time under SLEARN and 16.92% under 3Sigma. This happens
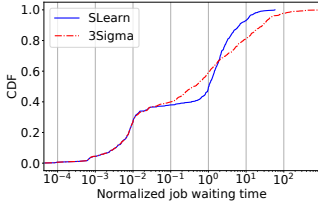
Figure 6: CDF of waiting times for wide jobs in GTrace11.
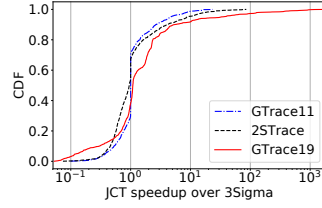
Figure 7: [Testbed] CDF of speedup: SLearn vs 3Sigma.
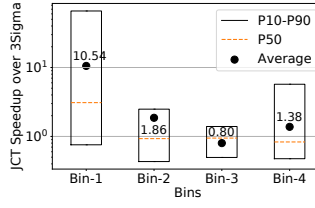
Figure 8: Performance breakdown into the bins in Table 8.
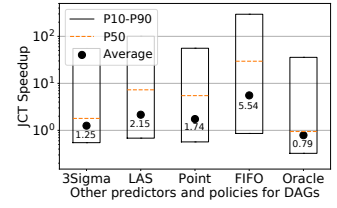
Figure 9: JCT speedup using SLearn-DAG over baselines for GTrace19-DAG.

Table 8: Breakdown of jobs based on total duration and width (number of tasks) for 2STrace. Shown in brackets are a bin's fraction of all the jobs in the trace in terms of job count and total job runtime.

	width < 3 (thin)	width ≥ 3 (wide)
size $< 10^3 s$ (sm)	bin-1 (4.55%, 0.01%)	bin-2 (28.73%, 0.06%)
size $\geq 10^3 s$ (lg)	bin-3 (14.29%, 5.41%)	bin-4 (52.43%, 94.52%)

because when the system load is relatively high, although jobs incur the sampling delay under SLearn, they also experience queuing delay under 3Sigma, and the more accurate prediction of SLearn allows them to be scheduled following Shortest Job First more closely than under 3Sigma.

A detailed analysis of how the system load of the trace affects the relative job performance under the two predictors can be found in the Appendix in [42].

5.2.6 Testbed Experiments

We next perform end-to-end evaluation of SLearn and 3Sigma on our 150-node Azure cluster. Fig. 7 shows the CDF of JCT speedups using SLearn over 3Sigma using 2STrace, GTrace11 and GTrace19. SLearn's performance on the testbed is similar to that observed in the simulation. In particular, SLearn achieves average JCT speedups of 1.33×, 1.46×, and 1.25× over 3Sigma for the 2STrace, GTrace11, and GTrace19 traces, respectively.

5.2.7 Binning Analysis

To gain insight into how different jobs are affected by SLearn over 3Sigma, we divide the jobs into four bins in Table 8 for 2STrace and show the JCT speedups for each bin in Fig. 8. The results for the other two traces are similar and are omitted due to page limit.

We make the following observations. (1) SLearn improves the JCT for 82.46% of the jobs in Bin-1 and the average JCT speedup for the bin is 10.54×. This happens because the jobs in this bin are thin and hence SLearn assigns them high priorities, which is also the right thing to do since these jobs are also small. (2) For bin-2, SLearn achieves an average JCT speedup of 1.86× from better prediction accuracy of SLearn. The speedups are lower than for Bin-1 as the jobs have to undergo sampling. However, Bin-1 and Bin-2 make up only 0.01% and 0.06% of the total job runtime and thus have little impact on the overall JCT. (3) Bin-3, which has

Table 9: Sensitivity analysis for thinLimit. Table shows average JCT speedup over 3Sigma.

thinLimit	2	3	4	5	6
2STrace	1.23x	1.28x	1.14x	0.97x	0.84x
GTrace11	1.54x	1.56x	1.55x	1.54x	1.53x
GTrace19	1.33x	1.32x	1.32x	1.30x	1.29x

14.29% of the jobs and accounts for 5.41% of the total job size, has a slowdown of 20.00%. The main reason is that SLearn treats thin jobs in the FIFO order, whereas 3Sigma schedules them based on predicted sizes. (4) Bin-4, which accounts for a majority of the job and total job size, has an average speedup of 1.38×, which contributes to the overall speedup of 1.28×. The job speedups come from more accurate job runtime estimation of SLearn over 3Sigma. Finally, we note that while for the 2Sigma trace, the majority of thin jobs are large, for the Google 2011 (Google 2019) trace, only 1.90% (1.60%) of the total number of jobs are thin and large and they make up only 0.5% (0.5%) of the total job runtime..

5.2.8 Sensitivity to Thin Job Bypass

Finally, we evaluate SLearn's sensitivity to thinLimt. Table 9 shows that for GTrace11 and GTrace19, the average JCT speedup barely varies with thinLimit, but for 2STrace, there is a big dip when increasing thinLimit to 4 or 5. This is because a significant number of jobs in 2STrace have width 4, which causes the number of thin jobs to increase from 18.84% to 58.50% when increasing thinLimit from 4 to 5.

6 Scheduling for DAG Jobs

In earlier sections, we have focused on the benefits of sampling-based prediction. On the other hand, we envision that there are situations where it would be beneficial to combine sampling-based and history-based predictions. Below, we present our preliminary work applying such a hybrid strategy for scheduling DAG jobs. We will discuss several other use cases of a hybrid strategy in §7. Note that for multi-phase DAG jobs, simply applying sampling-based prediction to each phase in turn cannot estimate the whole DAG runtime ahead of time. Instead, our hybrid design below aims to learn the runtime properties and optimize the performance of a multi-phase DAG job *as a whole* (*e.g.,* [30, 33]).

Hybrid learning for DAGs (SLearn-DAG**).** The key idea

of SLearn-DAG is to adjust history-based prediction of the runtime of DAG jobs using sampling-based learning of its first stage. Upon arrival of a new DAG job, we estimate the runtime of its first stage using sampling-based prediction as described in §5.1.2, denoted as d_s. We also estimate the duration of this stage using history-base 3Sigma, denoted as d_h, and compute the adjustment ratio of $\frac{d_s}{d_h}$. For each of the remaining stages of the DAG, we predict their runtime using 3Sigma and then multiply it with the adjustment ratio. In a nutshell, this hybrid design reduces the error of history-based prediction due to staleness of the learning data, while avoiding the delay of sampling across all other stages.

History-based learning for DAGs (3Sigma-DAG). This is a straight-forward extension of 3Sigma. Upon arrival of a DAG job, it predicts independently the runtime for each stage using the 3Sigma and sums up the estimated runtime of all stages as the estimated runtime of the entire DAG.

We similarly extended other baselines described in §5.1.3 for DAG job.

Experimental setup. We evaluated SLearn-DAG against 3Sigma-DAG by replaying cluster trace in simulation experiments based on GS (§5.1.1). We kept the simulation setup and parameters the same as used in the other experiments. In particular, a DAG is placed in the corresponding priority queue based on its estimated total runtime.

DAG Traces. The only publically available DAG trace we could find is a trace from Alibaba [3], which could not be used as it does not contain features required for history-based prediction using 3Sigma. Instead, we followed the ideas in previous work, *e.g.,* Branch Scheduling [34], to generate a synthetic DAG trace of about 900 jobs using the Google 2019 trace [11], denoted as GTrace19-DAG. The number of stages in DAGs in the GTrace19-DAG was randomly choosen to be between 2-5 and each stage is a complete job from the Google 2019 trace. The jobs that are part of the same DAG have the same *jobname* and the same *username.*

Results. The results in Fig. 9 show that SLearn-DAG achieves significant speedup over other designs. The speedup is 1.26× over 3Sigma-DAG, 2.15× over LAS-DAG, and 1.74× over Point-Est-DAG. Looking deeper, we find that our sampling-based prediction still yields higher prediction accuracy: the P50 prediction error is 33.90% for SLearn-DAG, compared to 47.21% for 3Sigma-DAG. On the other hand, for DAG jobs the relative overhead of sampling (e.g, the delay) is lower since only the first stage is sampled. Together, they produce speedup comparable to earlier sections.

7 Discussions and Future Work

Combining history and sampling. In addition to improving the scheduling of DAG jobs (§6), we discuss several additional motivations for combining history- and sampling-based learning. (1) For workloads with both recurring and first-time jobs, sampling-based learning can be used to estimate properties for first-time jobs, while history-based learning can be used for recurring jobs. (2) When the workload has both thin and wide jobs, history-based learning can be used for estimating the runtime for thin jobs, while sampling-based learning is used for wide jobs. (3) History-based learning can be used to establish a prior distribution, and sampling-based approach can be used to refine the posterior distribution. Such a combination is potentially more accurate than using either approach alone. For example, knowing the prior distribution of task lengths can help to develop better max task-length predictors, which can be useful for jobs with deadlines. (4) Though not seen in the production traces used in our study, in cases when task-wise variation and job-wise variation fluctuate, adaptively switching between the two prediction schemes may also help. (5) When the cluster is heterogeneous, an error adjustment using history, similar to what we did in §6, can be applied.

Dynamic adjustment of ThinLimit. ThinLimit is a subjective threshold. It helps in segregating jobs for which waiting time due to sampling overshadows the improvement in prediction accuracy. The optimal choice of this limit will depend on the cluster load at the moment and hence can be adaptively chosen like the sampling percentage (Fig. 3 on page).

Heterogeneous clusters. Extending sampling-based learning to heterogeneous clusters requires adjusting the task sampling process. One idea is to schedule pilot tasks on homogeneous servers and then scale their runtime to different types of servers using the ratio of machine speeds.

8 Conclusions

In this paper, we performed a comparative study of task-sampling-based prediction and history-based prediction commonly used in the current cluster job schedulers. Our study answers two key questions: (1) Via quantitative, trace and experimental analysis, we showed that the task-sampling-based approach can predict job runtime properties with much higher accuracy than history-based schemes. (2) Via extensive simulations and testbed experiments of a generic cluster job scheduler, we showed that although sampling-based learning delays non-sampled tasks till completion of sampled tasks, such delay can be more than compensated by the improved accuracy over the prior-art history-based predictor, and as a result reduces the average JCT by 1.28×, 1.56×, and 1.32× for three production cluster traces. These results suggest task-sampling-based prediction offers a promising alternative to the history-based prediction in facilitating cluster job scheduling.

References

[1] 2sigma hedge fund. www.twosigma.com.

[2] 2sigma's proprietary job scheduler. https://www.twosigma.com/insights/article/cook-a-fair-preemptive-resource-scheduler-for-compute-clusters/.

[3] Alibaba cluster trace. https://github.com/alibaba/clusterdata.

[4] Apache hadoop. http://hadoop.apache.org.

[5] Apache hadoop yarn. https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[6] Apache hive. http://hive.apache.org.

[7] Apache spark. http://spark.apache.org.

[8] Cluster trace from google - 2011. https://github.com/google/cluster-data/blob/master/ClusterData2011_2.md.

[9] A document released by google containing schema and details of the cluster trace released by google. https://drive.google.com/open?id=0B5g07T_gRDg9Z0lsSTEtTWtpOW8.

[10] Dss scheduler. https://github.com/epfl-labos/DSS.

[11] Google cluster-usage traces, retrieved 21st july 2020. https://research.google/tools/datasets/google-cluster-workload-traces-2019/.

[12] Google cluster-usage traces, retrieved 21st july 2020. https://drive.google.com/file/d/10r6cnJ5cJ89fPWCgj7j4LtLBqYN9RiI9/view.

[13] Hadoop patch for numeric histogram. https://issues.apache.org/jira/browse/YARN-2672.

[14] Microsoft azure. http://azure.microsoft.com.

[15] A patch for gridmix. https://issues.apache.org/jira/browse/YARN-2672.

[16] Personal communication with a 2sigma engineer regarding properties of the 2sigma trace used.

[17] A private trace collected by 2sigma engineers from their clusters. www.twosigma.com.

[18] Resutls on the posteriro distribution with gaussian priors. https://people.eecs.berkeley.edu/ jordan/courses/260-spring10/lectures/lecture5.pdf.

[19] Faraz Ahmad, Srimat T. Chakradhar, Anand Raghunathan, and T. N. Vijaykumar. Shufflewatcher: Shuffle-aware scheduling in multi-tenant mapreduce clusters. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 1–13, Philadelphia, PA, 2014. USENIX Association.

[20] George Amvrosiadis, Jun Woo Park, Gregory R. Ganger, Garth A. Gibson, Elisabeth Baseman, and Nathan DeBardeleben. On the diversity of cluster workloads and its impact on research results. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 533–546, Boston, MA, 2018. USENIX Association.

[21] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 285–300, Broomfield, CO, 2014. USENIX Association.

[22] Ronnie Chaiken, Bob Jenkins, Per-AAke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. Scope: Easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276, August 2008. http://dx.doi.org/10.14778/1454159.1454166.

[23] Mosharaf Chowdhury and Ion Stoica. Efficient coflow scheduling without prior knowledge. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 393–406, New York, NY, USA, 2015. ACM.

[24] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient coflow scheduling with varys. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 443–454, New York, NY, USA, 2014. ACM.

[25] Andrew Chung, Jun Woo Park, and Gregory R. Ganger. Stratus: Cost-aware container scheduling in the public cloud. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, pages 121–134, New York, NY, USA, 2018. ACM.

[26] Edward G Coffman and Leonard Kleinrock. Feedback queueing models for time-shared systems. *Journal of the ACM (JACM)*, 15(4):549–576, 1968.

[27] Carlo Curino, Djellel E. Difallah, Chris Douglas, Subru Krishnan, Raghu Ramakrishnan, and Sriram Rao. Reservation-based scheduling: If you're late don't blame us! In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 2:1–2:14, New York, NY, USA, 2014. ACM.

[28] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.

[29] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. Kairos: Preemptive data center scheduling without runtime estimates. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, pages 135–148, New York, NY, USA, 2018. ACM.

[30] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 99–112, New York, NY, USA, 2012. ACM.

[31] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 323–336, Berkeley, CA, USA, 2011. USENIX Association.

[32] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 455–466, New York, NY, USA, 2014. ACM.

[33] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 65–80, Savannah, GA, 2016. USENIX Association.

[34] Zhiyao Hu, Dongsheng Li, Yiming Zhang, Deke Guo, and Ziyang Li. Branch scheduling: Dag-aware scheduling for speeding up data-parallel jobs. In *Proceedings of the International Symposium on Quality of Service*, IWQoS '19, New York, NY, USA, 2019. Association for Computing Machinery.

[35] Zhe Huang, Bharath Balasubramanian, Michael Wang, Tian Lan, Mung Chiang, and Danny HK Tsang. Need for speed: Cora scheduler for optimizing completion-times in the cloud. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 891–899. IEEE, 2015.

[36] Calin Iorgulescu, Florin Dinu, Aunn Raza, Wajih Ul Hassan, and Willy Zwaenepoel. Don't cry over spilled records: Memory elasticity of data-parallel applications and its application to cluster scheduling. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 97–109, Santa Clara, CA, 2017. USENIX Association.

[37] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[38] Akshay Jajoo, Rohan Gandhi, and Y. Charlie Hu. Graviton: Twisting space and time to speed-up coflows. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, Denver, CO, 2016. USENIX Association.

[39] Akshay Jajoo, Rohan Gandhi, Y. Charlie Hu, and Cheng-Kok Koh. Saath: Speeding up coflows by exploiting the spatial dimension. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '17, pages 439–450, New York, NY, USA, 2017. ACM.

[40] Akshay Jajoo, Y. Charlie Hu, and Xiaojun Lin. Your coflow has many flows: Sampling them for fun and speed. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 833–848, Renton, WA, 2019. USENIX Association.

[41] Akshay Jajoo, Y. Charlie Hu, and Xiaojun Lin. A case for flow sampling based learning for coflow scheduling, 2021. http://arxiv.org/abs/2108.11255.

[42] Akshay Jajoo, Y. Charlie Hu, Xiaojun Lin, and Nan Deng. A case for task sampling based learning for cluster job scheduling, 2021. http://arxiv.org/abs/2108.10464.

[43] Virajith Jalaparti, Peter Bodik, Ishai Menache, Sriram Rao, Konstantin Makarychev, and Matthew Caesar. Network-aware scheduling for data-parallel jobs: Plan when you can. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 407–420, New York, NY, USA, 2015. ACM.

[44] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Inigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. Morpheus: Towards automated slos for enterprise clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 117–134, Savannah, GA, 2016. USENIX Association.

[45] Shonali Krishnaswamy, Seng Wai Loke, and Arkady Zaslavsky. Estimating computation times of data-intensive applications. *IEEE Distributed Systems Online*, 5(4):1 – 12, 2004.

[46] Misja Nuyens and Adam Wierman. The foreground–background queue: a survey. *Performance evaluation*, 65(3-4):286–307, 2008.

[47] Jun Woo Park, Alexey Tumanov, Angela Jiang, Michael A. Kozuch, and Gregory R. Ganger. 3sigma: Distribution-based cluster scheduling for runtime uncertainty. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 2:1–2:17, New York, NY, USA, 2018. ACM.

[48] Idris A. Rai, Guillaume Urvoy-Keller, and Ernst W. Biersack. Analysis of las scheduling for job size distributions with high variance. In *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '03, pages 218–228, New York, NY, USA, 2003. ACM.

[49] Kaushik Rajan, Dharmesh Kakadia, Carlo Curino, and Subru Krishnan. Perforator: Eloquent performance models for resource optimization. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 415–427, New York, NY, USA, 2016. ACM.

[50] Warren Smith, Ian Foster, and Valerie Taylor. Predicting application run times using historical information. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 122–142, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[51] Alexey Tumanov, Angela Jiang, Jun Woo Park, Michael A. Kozuch, and Gregory R. Ganger. Jamaisvu: Robust scheduling with auto-estimated job runtimes. In *Technical Report CMU-PDL-16-104*. Carnegie Mellon University, 2016.

[52] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. Tetrisched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 35:1–35:16, New York, NY, USA, 2016. ACM.

[53] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[54] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, Carlsbad, CA, October 2018. USENIX Association.

[55] Yong Xu, Kaixin Sui, Randolph Yao, Hongyu Zhang, Qingwei Lin, Yingnong Dang, Peng Li, Keceng Jiang, Wenchi Zhang, Jian-Guang Lou, Murali Chintalapati, and Dongmei Zhang. Improving service availability of cloud systems by predicting disk error. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 481–494, Boston, MA, 2018. USENIX Association.

[56] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 265–278, New York, NY, USA, 2010. ACM.

Starlight: Fast Container Provisioning on the Edge and over the WAN

Jun Lin Chen
University of Toronto

Daniyal Liaqat
University of Toronto

Moshe Gabel
University of Toronto

Eyal de Lara
University of Toronto

Abstract

Containers, originally designed for cloud environments, are increasingly popular for provisioning workers outside the cloud, for example in mobile and edge computing. These settings, however, bring new challenges: high latency links, limited bandwidth, and resource-constrained workers. The result is longer provisioning times when deploying new workers or updating existing ones, much of it due to network traffic.

Our analysis shows that current piecemeal approaches to reducing provisioning time are not always sufficient, and can even make things worse as round-trip times grow. Rather, we find that the very same layer-based structure that makes containers easy to develop and use also makes it more difficult to optimize deployment. Addressing this issue thus requires rethinking the container deployment pipeline as a whole.

Based on our findings, we present Starlight: an accelerator for container provisioning. Starlight decouples provisioning from development by redesigning the container deployment protocol, filesystem, and image storage format. Our evaluation using 21 popular containers shows that, on average, Starlight deploys and starts containers $3.0\times$ faster than the current state-of-the-art implementation while incurring no runtime overhead and little (5%) storage overhead. Finally, it is backwards compatible with existing workers and uses standard container registries.

1 Introduction

Docker and other container engines are a popular approach for software provisioning due to their low overhead, standardization, and ease of use [3, 41, 53, 60]. They provide isolation and standardized packaging for application files, and are supported by a large suite of standard tools [16, 18, 21, 23, 24]. Unlike VMs, containers are lightweight and easy to update: even lightweight VMs [1, 38] require re-building and re-deploying the entire image. Container images, on the other hand, are built as a stack of layers; updating a component can be as simple as rebuilding its layer rather than the entire image [15].

Similarly, we can extend a container by adding layers to the top of its stack. Deploying is also straightforward: fetch compressed layers from a *registry* server such as Docker Hub, decompress them, mount using a layered filesystem [34], and start the container process. The stack-of-layers structure thus makes containers easy to develop and maintain, and fits well into modern development workflows [5].

Though originally designed to be used inside a cloud datacenter [20], containers are becoming increasingly popular in edge computing, mobile, and multi-cloud settings [11, 22, 25, 47, 56, 61].[1] Placing workers outside the cloud and closer to the user brings many advantages such as lower latency, bandwidth and power reduction, and privacy [52, 59]. Containers can be used to provision network functions at mobile base stations [13], Function-as-a-Service (FaaS) runtimes on local datacenters [43], local replicas in distributed stores [42], or components of distributed applications [61].

However, as systems grow larger and more complex, fast container provisioning is increasingly important. For example, Container-as-a-Service (CaaS) and FaaS providers must be able to provision workers quickly [3, 41, 60]. Another common case is rolling software updates, where we must update software across many thousands of workers [6, 50]. Edge computing brings its own set of challenges: high latency upstream links, bandwidth limits, resource-constrained local datacenters and workers, and user mobility. Pulling container images from a registry in the cloud to an edge worker takes a long time over wide-area links [25]. Another issue is user mobility, which causes frequent reconfigurations [57], making worker provisioning a common operation. Finally, limited resources in edge datacenters means that placing a local registry or cache at every edge can be expensive [25].

While there is work on improving container provisioning time, many are designed for the cloud [25, 60, 63], and are ill-suited for edge computing scenarios. For example, FaaS-Net [60] uses a tree of workers to deploy containers in parallel, which is infeasible when latency is large and bandwidth

---

[1] The distinctions between these settings are not relevant for this work, hence we will refer to all of these using the umbrella term "edge computing".

is limited. Another popular approach is on-demand download [28,37,58], where we start containers early and download files on demand. These scale poorly with even moderate latency, even though many containerized applications do not necessarily require all mounted files immediately.

Our Contributions We identify three barriers to fast container provisioning. First, the layer-based structure that makes containers so convenient also prevents effectively applying common optimizations such as eliminating redundancy and downloading files on-demand. Second, the pull-based design of current approaches, where workers request what they need, becomes detrimental as latency grows. Finally, current approaches do not explicitly address the common scenario of software updates. We argue that faster provisioning requires a holistic approach to container deployment.

Motivated by these insights, we present **Starlight: an accelerator for provisioning container-based applications** that decouples the mechanism of container provisioning from container development. Starlight maintains the convenient stack-of-layers structure of container images, but uses a different representation when deploying them over the network. The development and operational pipelines remain unchanged: users can use existing containers, tools, and registries. In designing Starlight, we revisit every aspect of the container deployment mechanism:

- A redesigned **worker-cloud deployment protocol** sends all file metadata first, allowing containers to start before file contents are available. It uses a push-based approach to avoid costly round-trip requests: workers can specify what they already have in store, so we send only the files they need in the order they would be needed.
- On the worker side, we use a **new filesystem** to mount files as soon as metadata is available, allowing our **custom *snapshotter plugin*** to start containers quickly while downloading file contents in the background. When a container opens a file whose contents are pending, we block until the contents are available.
- Workers connect to a new **proxy** component in the cloud which implements the new protocol. The proxy optimizes the list and order of files on-demand, across multiple layers and containers This reduces duplication and makes updates faster. The proxy works transparently with existing infrastructure: compressed layers are stored in a standard registry, and legacy workers can connect to that registry as normal.
- A seekable **compressed layer format** allows the proxy to send individual compressed files to the worker without having to decompress stored layers first. This format has low overhead (average of 4.2%) and is backwards compatible with existing workers and registries, so there is no need to store container images in two formats.

We use 21 popular container images to evaluate Starlight across a range a range of network latencies, bandwidths, and scenarios. Our results show that Starlight substantially outperforms other approaches across all latencies, with 3.0× faster provisioning than a state-of-the-art baseline [21], and 1.9× faster on average than the next best approach [58]. Starlight also improves provisioning inside the cloud; for example it can deploy updates 35% faster than prior work [58]. In fact, Starlight containers often start faster than the time it would take to merely download an optimized container image. Finally, Starlight has little-to-no runtime overhead: its worker performance matches the standard state-of-the-art approach.

Starlight is currently available as an open source project at https://github.com/mc256/starlight.

2 Background

A *container* is a process that is isolated from the host system using techniques such as cgroups and namespaces [35]. A container is structured as a stack of *layers*, where each layer contains a part of the filesystem tree for the containerized application. Layers are mounted by the container process using a filesystem such as OverlayFS [34] that presents the containerized application with a *merged view*: files in upper layers replace those in lower layers, making it easy to update container contents using copy-on-write from lower layers. Most layers are read-only; writes go to a top read-write layer using copy-on-write as needed. A *container image* is the set of files and associated metadata that represent the container at rest (i.e., when it is not running). Concretely, container images are comprised of container configuration metadata and a sequence of *compressed layers*: compressed files that store the files in the layer and their associated metadata.

Containers are easy to develop, maintain, and deploy, due to their layer-based structure and standardized tooling. For example, developers can build new containerized applications by adding layers on top of an existing container image; packaging application updates is similarly straightforward. This also makes security updates for underlying components fast and automatic: applying an update simply requires updating the base layer. The repository of container images (the *registry* server) thus resembles a tree where individual images are split off from a common point.

Containers also make software provisioning easy using a three-phase process managed by a *container engine* on the worker such as containerd [21] or Docker [18]: (i) **pull** the requested container image from the registry and decompress its layers, (ii) **create** a container instance by preparing an initial snapshot of its filesystem state, and (iii) **start** the container instance, which involves mounting the snapshot filesystem and starting the container process using a standard *runtime*.

2.1 Edge Computing

Edge computing, defined broadly in this work, is the idea of placing computing resources outside the cloud, closer to the data or end users [52,62]: near the network edge (e.g., local
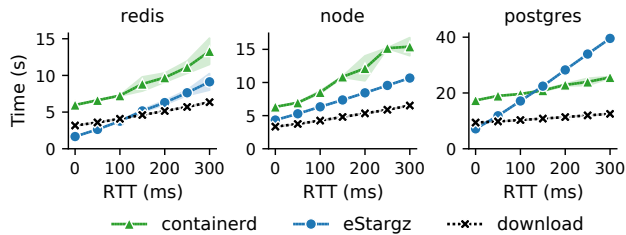
Figure 1: Mean container provisioning time across a range of latencies; shaded area show standard deviation across 5 runs.

datacenter, base station), user devices (e.g., mobile phone), or even in low Earth orbit [14]. We include in this definition settings such as mobile computing, content delivery networks (CDNs), Internet of Things (IoT), wide-area networks, and multi-cloud deployments.

Edge computing provides many benefits. For example, the short distance to users and data means faster and more consistent response times. And, since we no longer need to send all data to the cloud for processing, it improves privacy and reduces bandwidth usage. Other benefits include robustness to network failures and emission reduction [51, 52, 54].

Computing on edge workers has its drawbacks, however. First, they have much higher round-trip times (RTT) to the cloud. Recent work has found RTT ranging from 10ms to 400ms [10, 59] on terrestrial internet, and medians of 45–724ms on satellite-based internet [40]. Cross-datacenter latencies are also high, with one cloud provider reporting RTT between 2 to 400ms [45]. Bandwidth is also limited, with inter-datacenter bandwidths of 30–250Mbps [48]. Second, unlike cloud datacenters that offer virtually endless compute and storage, edge data centers are typically resource-constrained [54]. This encourages aggressive repurposing of workers, which makes fast provisioning even more important. For example, maintaining a pool of "hot" workers for elasticity is common in the cloud FaaS infrastructures, but is more expensive on the edge [43]. Lastly, edge and mobile applications are more affected by user mobility than cloud applications: as users move the nearest edge datacenter changes, which entails more frequent reconfiguration [57], i.e., provisioning.

3   Motivation

To explore the effect of latency on containers, we use `containerd` [21] to provision three popular containers over a 100Mbps connection with variable latency (see §5.1 for technical details). Figure 1 shows provisioning time, defined as the time it takes for the containerized application to download, decompress, start, and be ready. For comparison, we also show the download time for a file of equivalent size (dashed lines). We observe that `containerd` time increases substantially as RTT grows, and can even triple when RTT is

300ms. Moreover, in all cases provisioning time increases at a faster rate than would be expected simply due to extra network latency, which can be seen by comparing provisioning time to download time.

We also compare to eStargz [58], a recent approach that accelerates provisioning by starting the container before its layers have finished downloading and retrieving individual files on-demand. Prior work has found that many files are not used during container startup [28], indeed our three example containers access less than 1% of their files during startup (comprising 1–39% of data). Rather than wait until all files are available, eStargz starts the container quickly and download files on-demand [27, 28, 65]. It goes further by optimizing the order of files in each compressed layer such that the "hot" files needed early in container startup are placed first, thus avoiding redundant requests. Workers first fetch the hot part of each layer, start the container, and continue fetching the remaining files in the background or lazily on-demand.

As Figure 1 shows, when latency is small eStargz can accelerate provisioning. However, as RTT grows eStargz scales worse than the baseline and can even become slower than baseline `containerd`, as demonstrated for `postgres` with RTT of 150ms or above.

In the rest of this section, we analyze what makes optimizing provisioning difficult. We find that the root cause for slow provisioning time is the overall design of the provisioning pipeline: it is pull-based, designed around the stack-of-layers abstraction container images, and does not explicitly consider container updates. We show below that this design hinders optimization effort – both on the edge and in the cloud.

3.1   Pull-based Protocol

The protocol used to deploy containers to workers is pull-based: workers simply download the compressed layers they need from the registry using HTTP requests. This straightforward design avoids redundant pulls of layers that the worker already has, and works well inside datacenters. However, outside the cloud this can cause queueing delays, since registry implementations limits the number of concurrent connections per client to 2 or 3 [17]. Most containers have more layers [28], so the resulting cumulative delay adds up as RTT grows. Increasing the maximum number of concurrent connections could overwhelm the registry and may be impractical for resource-constrained workers.

On-demand downloading further exacerbates queuing by making even more HTTP requests to the registry. eStargz [58] uses a filesystem file access trace to determine the file order in compressed layers. In practice, however, the file access order of container workloads is not entirely deterministic due to multi-threading and runtime configuration. Container startup is thus slowed as multiple HTTP request due to out-of-order file accesses queue in the registry and delay one another.[2]

---

[2]Interestingly, excessive round-trips and queuing delays were also ob-

3.2 Layered-based Structure

Container images are structured as a stack of independent layers: each layer is stored separately, and contains its own metadata (e.g., list of files). While convenient for development, we argue that this makes optimizing provisioning more difficult: first, the information on container contents is distributed across multiple layers; second, because layers are the wrong granularity for provisioning protocols; and third, layer reuse does not capture updates well.

Distributed Metadata The first issue is that file metadata, including the list of files in the container contents, is not sent separately as part of the container image. Rather, each compressed layer includes its own list of files, and their metadata is intermingled with file contents. Yet, we cannot start a container early since list of files in a container is unknown until all layers are retrieved.

Consider again eStargz: since standard container images lack file metadata, eStargz stores a table of contents (ToC) at the end of every layer. Unfortunately, neither the size of compressed layers nor the exact beginning of the ToC is encoded in the image metadata. This in turn means at least two and perhaps three HTTP requests per layer: one to determine the size of its compressed image file, another to retrieve the layer's ToC from an estimated position before the end, and potentially a third if the ToC is larger than expected.

Fixing this is not trivial, since container images are standardized; careless changes would make development harder. For example, adding a table of contents to container image metadata requires changing the standard and updating a huge number of existing tools used by developers [18, 21, 23, 46].

Layer vs. File Granularity Second, and perhaps counter-intuitively, the layer-based structure makes deployment slower due to cross-layer (and cross-container) redundancy. Containers evolve one layer at a time by extending other images with new layers. To update a file, we first copy it from the original read-only layer to the top read-write layer. Changing file metadata (e.g., ownership) also requires copying since layers cannot refer to each other. In both cases the original file remains in the previous layer, with no indication that this has happened. This cross-layer data duplication cannot be captured explicitly since file metadata is stored in the layers, and cannot be exploited by compression since layers are compressed independently.

Table 1 illustrates the cost of such redundancy for our sample containers by comparing the required download size using the baseline layer-based approach, to the size of an optimized "delta" update that only includes changed files and removes duplicates across layers.[3] The inflation in update sizes ranges from 1.23× (redis) to a whopping 10.54× (node). Indeed, a

Container	From → To	Baseline	Delta
redis	6.2.1 → 6.2.2	9.6	7.8
node (alpine)	16-3.11 → 16-3.12	39.0	3.7
postgres	13.1 → 13.2	109.5	24.9

Table 1: Package size (MB) of standard and optimized update.

recent analysis of Docker Hub [64] found that 90% of layers are only referenced by a single image, but over 99.4% of files had duplicates. Exploiting this cross-layer duplication during provisioning is difficult since file metadata is distributed across multiple layer.

While there has been prior work that proposes deduplicating the registry [55, 63], this does not reduce provisioning time since (by design) the downloaded container images and provisioning protocol remain the same. Rather, such work focus on saving registry space.

Limited Layer-reuse Ideally, an updated container image would share common layers with its previous version, so deploying updates requires only fetching and decompressing the new layers. Unfortunately, even a minor change to a single layer low in the stack causes cascading effect where all layers above it must be updated, even though their contents are mostly identical [15]. On such example is updating a worker from `postgres:13.1` to `postgres:13.2`. These two container images share no layers since an update to the `debian:buster-20210208-slim` image forced an update to all downstream layers. Provisioning this update requires downloading and decompressing the entire image, even though the total size of changed files is much smaller. Our analysis of 21 popular containers (Table 2) suggests that layer reuse only captures 3% of duplication, on average.

3.3 No Explicit Update Support

Provisioning a worker is not a rare operation. Rather, over the lifetime of a worker, we will deploy containers many times and for different reasons: initial provisioning, software updates, security patches, and so on. This even more common on edge workers due to user mobility and limited resources at edge datacenters (§2.1). This not only results in frequent provisioning, but also means that worker contents is highly diverse: as workers get updated and repurposed, the version of the container image available in local storage varies from worker to worker. As discussed above, such updates are an opportunity for optimization since many of the files have not, in fact changed (§3.2).

However, the current design of the provisioning pipeline does not allow users to express update operations explicitly.

---

served in mobile web browsers that use HTTP/2 [36]. The underlying causes, however, are quite different (handshaking and packet losses, respectively). Determining whether the mitigation approaches in QUIC are applicable for container provisioning (or vice versa) is beyond the scope of this work.

---

[3]Flattening container images down to a single merged layer this way would mitigate many of the issues we discuss. However, would also eliminate the advantages of containers in the first place (§2), and would require an optimized image for every potential update path [47, 55].

Under the current approach, updates are treated as any other deployment: the worker simply pulls the needed layers from the registry. Depending on the update, this may or may not result in faster provisioning. While in theory we could prepare optimized provisioning packages in advance, the diversity in worker contents makes this approach impractical. A better approach is to compile the provisioning package dynamically, on-demand, by taking into account what is already available at the worker when selecting which files to include. Doing so, however, requires a capacity to express worker updates, which the current provisioning protocol does not support.

4 Starlight

Starlight is designed to accelerate container provisioning by considering the deployment pipeline holistically. In designing Starlight, we set out to achieve several goals. First, accelerate deployment on both low and high-latency links, and scale gracefully as latency grows. Second, preserve the advantages of containers for application developers. For the same reason, Starlight should be easy to adopt incrementally, without causing interference or requiring abrupt changes to working systems – Starlight should be backwards-compatible with non-Starlight workers, work with existing infrastructure, and have low overhead. Finally, Starlight should better support the common scenario of container updates.

4.1 Design Considerations

Starlight's design is driven by four key principles, informed by our analysis of container provisioning (§3): (1) start containers early, (2) send workers only what they need, (3) use a push-based design to avoid costly round-trips, and (4) prioritize worker performance over cloud effort. These lead to the following design decisions:

- The provisioning protocol should not resemble the stack-of-layer structure of container images. Instead, it should be pushed-based, and operate at file rather than layer granularity. The list and order of files should be jointly optimized across multiple layers and containers.
- The provisioning protocol should cleanly separate file metadata from contents, and send the metadata first. This allows Starlight to start containers early by mounting a "mock" filesystem while downloading contents in the background.
- The provisioning protocol should let workers explicitly request updates and specify what is available to them locally.
- Avoid changing registry by placing a *proxy* located near it, which lets us to change provisioning protocol without affecting existing workers.
- The proxy should create provisioning packages on-demand based on what the worker already has available. This makes updating workers more efficient, avoids inflating the registry with packages for every conceivable update, and places
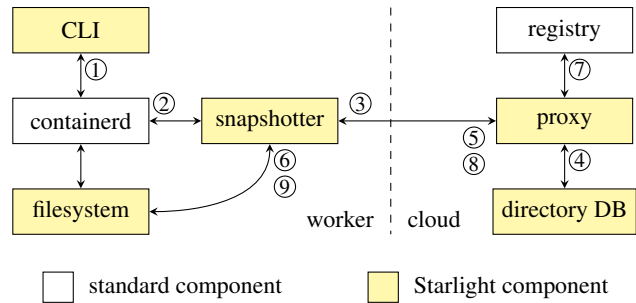
Figure 2: Starlight architecture.

the computational burden on the cloud where it is cheaper (§2.1). Supporting this requires storing a table of contents and file metadata for every container.
- Storing compressed layers using a seekable backwards-compatible format allows both Starlight and legacy workers to use the same compressed layer files and standard registries, and avoids inflating the registry size.
- Use standard container images to support the large ecosystem of existing tools for building, storing, and serving containers [16, 18, 21, 23, 24, 30, 46].

4.2 Overview

Figure 2 shows Starlight's architecture, which is comprised of a *proxy* and a *directory database* (§4.4) in the cloud next a standard registry; and a *snapshotter* plugin (§4.5) on the worker. The proxy and the snapshotter plugin communicate using the Delta Bundle Protocol (§4.3). The snapshotter plugin manages the lifecycle of the *filesystem* (§4.6) for the container instance. We also include a command line tool.

We first describe Starlight's operation at a high level, and how it maps to the three-step PULL-CREATE-START process.

Once the user issues a worker PULL command to deploy a container ①, the command is received by the standard containerd daemon. containerd then forwards the command to the Starlight *snapshotter* daemon ②, and waits for confirmation that the requested images have been found. The Starlight snapshotter opens an HTTPS connection to the Starlight *proxy* and sends the list of requested containers as well as the list of relevant containers that already exist on the worker ③. The proxy queries the *directory database* ④ for the list of files in the various layers of the requested container image, as well in the image already available on the worker. The proxy will then begin computing the *delta bundle* that includes the set of distinct compressed file contents that the worker does not already have, specifically organized to speed up deployment; In the background, the proxy issues a series of HTTPS requests to the registry ⑦ to retrieve the compressed contents of files needed for delta bundle. Once the contents of the delta bundle has been computed, the proxy creates a *Starlight manifest* (SLM) – the list of file metadata,

container manifests, and other required metadata – and sends it to the snapshotter ⑤, which notifies `containerd` that the PULL phase has finished successfully.

We can then use the SLM execute the CREATE phase: configuring the container and the Starlight filesystem ⑥.

Starlight then proceeds to the START phase: it mounts then launch container instances. Note that even though the container instances have launched, at this point the worker does not have the contents of many files (or perhaps all, for new deployments). Such files are mounted based on metadata only; when opened, the Starlight filesystem will block until file contents have arrived from the proxy. The proxy streams file contents back to the snapshotter ⑧, which in turn updates the filesystem to unblocks the access of the file ⑨. To optimize provisioning times, the proxy sends files in the order they will (likely) be needed; this order is determined when building the delta bundle, and relies on preprocessed information stored in the directory database.

4.3 Delta Bundle Protocol

Starlight uses a novel Delta Bundle Protocol to send container images to the worker. This single-request HTTP-based protocol is designed to reduce unnecessary transfers, avoid round-trips, and prioritise information needed to start the container. Much of Starlight's design is informed by the need to support the Delta Bundle Protocol.

A provisioning request includes the name and version of the requested the container image as part of the request URL, and optionally the old version in the worker's local storage. The response consists of two parts: a header and a body.

The Header   The header contains of all the information needed to start container instances of the requested images. It it comprised of (1) a Starlight Manifest (SLM), (2) a table of all layers digests from both the existing and requested container images, and (3) other data required by the implementation such as protocol version and authentication..

An SLM includes a standard Open Container Initiative (OCI) container image manifest file [24, 39], an OCI configuration file for the instance, a list of indices into the table of layer digests, and the filesystem table of content (ToC).

The ToC presents a merged (flattened) view of the requested container's filesystem (§2), providing sufficient information for the worker to mount the container's filesystem using StarlightFS without waiting for the response body. Each entry in the ToC includes the file name and path, type (e.g. regular file, link, or directory), attributes (e.g. ownership and timestamps), and an SHA256 hash of the file content. Additionally, every entry includes an index to the shared layer digests table in the delta bundle header, which enables reusing file contents on the worker's local storage. Finally, each entry also has an offset field which points to the file's *payload* – compressed file content – in the body of the delta bundle.

Using the SLM   The name, metadata, offset, and index into the digest list allow workers to reconstruct the requested container's filesystem. For new or updated files – those that the worker does not already have in its local storage – the offset points to the payload. This allows multiple file entries to reuse the same payload in the body of delta bundle, reducing the transfer volume. Alternatively, if a file's metadata has changed (e.g., ownership), the payload already exists on the worker. The ToC entry thus contains the new metadata, an empty payload offset, and an index pointing to the original layer in the list of digests.

The Delta Bundle Body   The body is a sequence of payloads (compressed file contents) for new or updated files, sent in the order which they are likely to be accessed. Since the header allows multiple file to reference the same payload – all payloads in the body are unique.

4.4 Proxy and Directory Database

Despite the name, the Starlight proxy is not merely a proxy server or a simple bridge. It is in charge of optimizing and building the delta bundle sent to the workers, as well as collecting and analyzing filesystem traces used in this optimization.

The Directory Database   The directory database stores the table of contents and file metadata for each container image in the registry, as well as additional information used by the proxy to compute and optimize the delta bundle.

Whenever a new container image is uploaded to the registry (triggered manually or by hooks), the proxy captures file metadata from all layers, generates the ToC for the merged view of the image, and then save the ToC, container manifest, and image configuration file to the directory database.

The ToC in the directory database is the same as the ToC included in the SLM with additional fields that fascilitate building the delta bundle body. First, it records the source compressed layer file, payload offset, and size for each file to help retrieve it from the registry. Second, it includes two extra columns, rank sum and hit count, used when sorting payloads; we discuss these below.

Trace Collection   To sort payloads in the order that the worker is likely to access, Starlight collects filesystem traces from the worker to analyse the file usage. Trace collection is identical to running a container until it reports it is ready. When initiated by the user, the worker starts the container image locally using a special mode of the Starlight filesystem (§4.6) that collects file accesses. The worker then uploads the trace to the proxy, which ranks all files in the trace according to their access order. Finally, for each file in a container image, the proxy increases its hit count by one and adds its rank to the rank sum column. The average rank of a file can be computed from its rank sum and count.

Since file access can be non-deterministic, our design supports multiple collection runs. Collecting one trace usually

takes up to 2 minutes per run, depending on the container. By default, we collect 10 traces for each container.

Note that while prior work [37, 58] stores file order information per layer inside the compressed layers, Starlight associates this information with a container image and stores it outside the registry. This provides several benefits. First, it is possible to update file usage without rebuilding the compressed layers. Second, it allows for the likelihood that a file in a layer used by different containers to be accessed differently during startup. Third, new container image can reuse the traces from a previous version – solving the cold start problem. Finally, it allows for future development such as adjusting payload orders online based on provisioning feedback.

Provisioning Process A provisioning request from a worker contains the names and tags of two images: the image requested for deployment, denoted by R, and the old version of the image in its local storage (assuming there is one) denoted by A. To build the delta bundle, the proxy first retrieves metadata from the directory database for both container images R and A. It then issues a series of asynchronous requests to the registry to retrieve the compressed layers for R. These will be used to construct the body of the delta bundle.[4] It then proceeds to prepare an optimized delta bundle header and send it to the worker. Once all requested layers to arrive from the registry, the proxy send the delta bundle body: for each payload in the delta bundle body as determined by the header, we copy compressed file content directly from the compressed layer and send them to the worker.

Optimizer The optimizer is responsible for selecting which compressed file content (payload) should be included in the body of the delta bundle and in which order, and then building the delta bundle header. Crucially, the optimizer does not require retrieving the compressed layers; the directory database contains all necessary information to build the delta bundle header. The optimization proceeds in several phases:

- **Merge:** load the merged (flattened) ToC for R and A from the directory database, denote them T_R and T_A.
- **Difference:** Compute the set difference $T' = T_R \setminus T_A$: for every file f in T_R, we look for a corresponding entry f' in T_A with the same hash and name. If we find one, we update the source layer index for f in T_R to the corresponding one in the old the entry in T_A update its source layer index to the corresponding layer of f'. This step takes $O(|T_R| + |T_A|)$ time and $O(1)$ space.
- **Consolidate:** Consolidate files in T' with the same payload. Assuming the chance of hash collision is low, this step takes $O(|T_R|)$ time and $O(|T_R|)$ space.

---

[4]Our current implementation retrieves entire compressed layers. This does not substantially affects provisioning time since the registry and the proxy are located in the same cloud datacenter. Nevertheless, we stress that Starlight's directory database and the seekable image format support retrieving only the contents of compressed files, by issuing HTTP range requests to the registry when building the delta bundle body. We are planning to implement this optimization in the immediate future.

- **Select:** Remove from T' files already available on the worker (whose source layer is in T_A).
- **Sort:** Sort the payloads in order of increasing average rank, $O(|T'| \log |T'|)$. If different files point to the same payload due to previous steps, use the lowest rank.

Compressed Layer Format The current format used to store compressed layers is the tar gzip format: a sequence of concatenated files with interleaved headers for metadata (e.g., timestamps, ownership), compressed as one data stream. This format is non-seekable: extracting a specific file requires decompressing the entire compressed layer until we reach the file, which takes time.

eStargz [58] uses an alternative seekable compressed layer format that compresses files individually (or 4KB chunks of larger a file) and appends an index at the end of the compressed layer into the offsets of compressed files and chunks. To maintain backwards compatibility, each file includes tar headers and footers, so the tarball data stream remains unchanged. The result is an increase in the size of compressed layers due to the index at the end and the additional tar headers and footers. Furthermore, compression is less effective since file are compressed separately.

Our proposed format follows similar ideas, with three differences. First, we do not include an index at the end of the compressed layer, and instead use the directory database to store the table of contents. This not only reduces the overhead of our compressed layer format, but allows the proxy to build the delta bundle while fetching compressed layers in the background. Second, we do not need to split files into 4kb chunks since we retrieve files wholly, which simplifies our provisioning protocol and reduces the size of the ToC. Finally, since we do not need the metadata in tar headers and footers during provisioning, we do not include them as part of the compressed stream of file contents, which further reduces payload size. The overhead of Starlight's format is only 4.4% for containers in Table 2 comparable to eStargz (4.7%).

4.5 Snapshotter Plugin

The `containerd` snapshotter daemon manages the life cycle of a container filesystem: from downloading images to keeping track of changes in the container's mounted file system. We take advantage of the snapshotter plugin-based design [7] to write a snapshotter plugin to support Starlight provisioning. Figure 3 shows an overview of the Starlight snapshotter plugin, which includes two components: the *downloader* and *metadata manager*. The snapshotter also maintains the instances of the user space component of StarlightFS – one for every mounted container instance.

Delta Bundle Downloader The downloader is responsible for downloading the delta bundle from the proxy and decompressing the payloads to designated locations (if an image has been completely downloaded, it is not started). Once the downloader receives the delta bundle header, it saves the
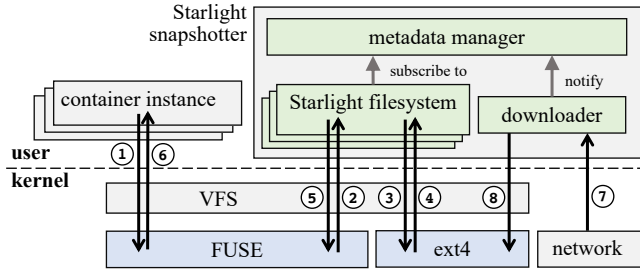
Figure 3: The flow of filesystem requests in Starlight.

SLM to the local storage and creates a metadata manager using the SLM. At this point, we have enough information to mount and start the container, so the snapshotter notifies the `containerd` daemon that the `PULL` phase has finished. In the background the downloader keeps receiving the payloads: it decompresses the payloads to its designated location according to the source layer information in the received SLM and the share layer digests in the delta bundle. Once the decompression of the payload has finished, it notifies the metadata manager. When a payload belongs to multiple files, the downloader creates hard links to avoid writing to the underlying filesystem multiple times.

Metadata Manager   Multiple container instances can start from the same container image. The metadata manager therefore acts as a centralize place for managing file's availability and its metadata. It maintains file metadata of all the files in a container image and manages files' actual location in the host file system. Most importantly, it manages the availability of file contents, and notifies StarlightFS once a file payload has decompressed. Once the worker has downloaded the entire image, we store its SLM locally so that future container instances launch from the local storage. When removing an old container image, the metadata manager removes any hard link references (if any) and copies the file to a new location if it is used by a newer version of this image.

4.6   The Starlight Filesystem (StarlightFS)

The customized filesystem serves two goals. First, we need to start containers early using only their SLM. Second, we want to reuse file contents across layers and images. As OverlayFS and other filesystems do not support both of these features, we use FUSE [33] to implement StarlightFS.

Structure   StarlightFS relies on the underlying host filesystem (e.g., ext4), similar to a typical OverlayFS and FUSE-OverlayFS. Like OverlayFS, StarlightFS provides a container with a merged view that combines multiple directories in the underlying filesystem that represent multiple read-only layers and a single read-write layer.

Starlight maintains a filesystem tree in memory, created from the merged view ToC in SLM. Each file (or directory) node keeps track of the actual location of the file contents –

whether it is in the read-only layer, in the read-write layer, or pending payload. As with the ToC, some nodes might reference read-only layers from the previous version of the container image (§4.3). Nodes of pending files will be notified by the metadata manager when the payload is available (in our implementation, by subscribing to a Go signal channel in the corresponding file entry of the metadata manager). The user space portion of StarlightFS is located in the snapshotter process to allow such low-overhead communication.

Note that StarlightFS does not maintain any file system state on its own, nor does it have any on-disk structures. Metadata for files in read-only layers is stored in the ToC. State for files in the read-write layer (i.e., mutable state) is stored in the underlying host filesystem, with changes forwarded to it immediately. For example, if a file is deleted by the container, we write a `whiteout` entry to the read-write layer, similarly to OverlayFS [34]. In case of a crash, error or remount, the tree and all other state are rebuilt using the saved SLM and underlying filesystem.

Operation   When starting a container instance, the snapshotter creates a filesystem instance which builds a filesystem tree from the metadata manager's ToC for this container image.

Figure 3 shows the flow of operations in StarlightFS. When a container instance performs a file operation, it is forwarded to StarlightFS via FUSE ①②. In the best scenario, the content of the file is already in the local filesystem (e.g., ext4 in Figure 3). Starlight uses the file path provided by the file node to opens the underlying file ③④ and then return the file handle back to the container instance ⑤⑥. In case the file contents are still pending, but the operation only involves reading metadata (e.g. `GETATTR`), StarlightFS returns the metadata immediately using the information in the file node.

When an operation on a pending file involves setting metadata (e.g. `SETATTR`) or accessing file content (e.g. `OPEN`, `FSYNC`), StarlightFS blocks the operation until the file is ready by subscribing to a Go signal channel associated with the file's ToC entry in the medtadata manager. Once the downloader has extracted the file payload ⑦⑧, it notifies the corresponding entry in the metadata manager, which closes the channel associated with the file's ToC entry. This releases any filesystem tree nodes that are waiting for the payload, while newly created instances will not be able to subscribe to a closed message channel. StarlightFS can then load the file from the local storage and update the file metadata if necessary ③④, then return the file to the container instance ⑤⑥. If this requires changing the file metadata or content, this file will be copied from the read-only layer to the read-write layer. All subsequent requests will be forwarded to the read-write layer.

5   Evaluation

We use 21 popular container images to evaluate Starlight's performance in both controlled and real-world networks. Our

main metric is *provisioning time*, defined as the time from the initial command to deploy a container on a worker, to the time the containerized application reports it is ready (as with HelloBench [28], this is determined by monitoring the application's `stdout`). To show the benefit of Starlight, we define two types of provisioning: a *fresh* deployment means the container worker does not have any prior images in its local storage, while an *update* means deploying the next available version of a container to a worker that already has the previous version deployed.

5.1 Experimental Setup

We use AWS EC2 to run our experiments. Container workers use `m5a.large` instances (AMD EPYC 7000 at 2.5GHz) with 2 cores (vCPUs) and 8GB RAM. The registry server runs Docker Registry 2.0[5] v2.7.1 on a `c5.xlarge` instance (Intel Xeon at 3.4GHz) with 4 vCPUs and 8GB RAM. The Starlight proxy and the metadata database run on a second `c5.xlarge` instance. All the machines run Ubuntu 20.04.3 LTS. We use Linux's Traffic Control tool [2,29] to control round-trip time and bandwidth between the worker and the other machines. Bandwidth is limited to 100Mbps unless otherwise specified. For cloud experiments, bandwidth and latency are not limited (RTT is ∼0.15ms). Each experiment is run 5 times.

Benchmark Approaches We compare Starlight to two state-of-the-art approaches: the `containerd` baseline [21] v1.5.0 and eStargz [8,58] v0.6.3.[6] The **baseline** implementation first downloads and decompresses all new compressed layers before launching the container. **eStargz** presorts the files in each compressed layer according their expected order of or use, and uses on-demand "lazy" download during deployment to handle for unexpected accesses: when a running container opens a file whose contents are not yet available, eStargz pauses the container and requests the file from the registry. We also plot two reference times: **warm startup** time denotes the container startup time once its image has already been downloaded and decompressed to local storage; **wget** time denotes the time to compute and download the Starlight delta bundle over the network, serving as a lower bound on provisioning time when not starting containers early.

Containers We evaluate Starlight on a variety of popular containers from Docker Hub [30]. Since many of the containers in the original HelloBench container suite [28] are outdated and can no longer be deployed using moderns tools, we instead take several of its most popular containers, finally, we add several container images used in edge computing applications. The full list of containers is available in Appendix A.1.

---

[5]This is the official Docker registry server [30,49].

[6]We do not compare to Slacker [28] as its source is not public and since eStargz is explicitly designed to supersede it in performance and features. Similarly, our preliminary experiments showed eStargz offers similar or superior performance to DADI [37].

5.2 Provisioning Time

Figure 4 shows the average normalized provisioning time for all the containers in Table 2 across a range of round-trip times (RTT) and network bandwidths. We normalize the provisioning time of each container to the time it takes to deploy a fresh worker using the baseline approach over a 100Mbps network with 0.15ms RTT. We also show the 95% confidence intervals to help establish statistical significance [12].

Our first immediate observation is that Starlight is the fastest provisioning approach for all latencies, bandwidths, and scenarios we study, except when provisioning fresh workers in the cloud, where Starlight provides similar performance to eStargz. It is significantly faster than both the state-of-the-art baseline approach and eStargz. Overall, Starlight provisioning is 3.0× faster on average than the baseline, and 1.9× faster than eStargz. Surprisingly, Starlight also frequently outperforms wget. In other words, Starlight early start design and effective scheduling of file payloads allows it to provision a fresh worker faster than the time it takes to merely download an optimized package. Conversely, eStargz, which also starts containers early, is on average slower than wget except when bandwidth is 54Mbps and RTT is low. Neither early start nor building optimized container images is sufficient in isolation; Starlight effectiveness is the result of its holistic design.

Effect of Latency When RTT is very low (i.e., inside a single datacenter), Both Starlight and eStargz are significantly faster than the baseline. However eStargz scales poorly when RTT grows due to its pull-based design that requests "out-of-order" files on-demand (§3). As latency grows, delays due to these requests add up: eStargz's provisioning time at RTT=300ms grows by 3.7× when going from RTT of 0ms to 300ms on a 500Mbps network. In comparison, the baseline provision time only doubles. For high bandwidth, high latency networks (e.g., satellite links) eStargz performance is close to the baseline approach, especially for updates.

Starlight, on the other hand, is far less sensitive to latency than the other approaches: its provisioning time grows at a slower rate than the baseline, eStargz, and wget. Starlight scales well not because its prediction of file access order is perfect (it is not), but rather due to its push-based design. Unlike eStargz, Starlight avoids flooding the registry with HTTP requests when containers open files "out of order", and instead waits for the file to arrive.

Deploying Updates Since updates are a common operation (§3.3), we also consider the provisioning time for updating containers on existing workers.

Starlight is very successful in optimizing updates: provisioning updates using Starlight (bottom row of Figure 4) is on average 1.7× faster than an equivalent fresh deployment (top row) using Starlight, and 2.5× faster than baseline fresh deployment.[7] The other approaches only show modest

---

[7]Harmonic mean of speedups across all bandwidths and latencies.
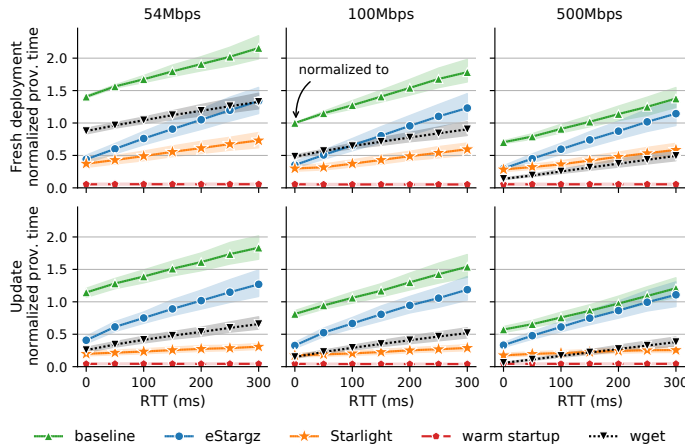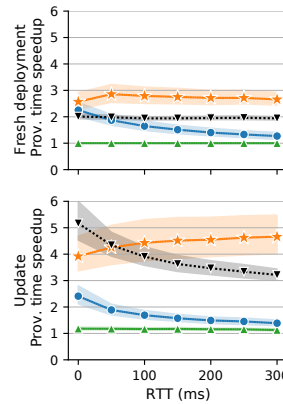
(a) Edge and WAN. (b) Speedup with 100Mbps. (c) Cloud time and speedup.
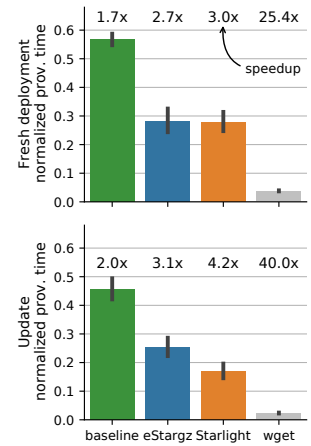
Figure 4: (a) Normalized provisioning time for different methods, round-trip times, and network bandwidth, aggregated across containers in Table 2. Solid line shows the geometric mean [19]; shaded areas show 95% confidence intervals. Top row shows fresh deployment, bottom row shows updates. Time is normalized to fresh deployment of the same container using the baseline approach with RTT of 0ms and a 100Mbps connection. (b) Speedups over baseline with 100Mbps (solid line shows harmonic mean). (c) Provisioning time and speedup in the cloud (RTT approximately ∼0.15ms, no bandwidth restriction).

improvement when provisioning updates: average update provisioning time for baseline and eStargz are close to those of fresh deployment. Additionally, we observe that Starlight update provisioning scales much better than the two other approaches as RTT grows. Finally, Starlight's transfer volume is smaller: the size of a median Starlight update is 30% that of a fresh update using the size of a baseline fresh deployment, while for eStargz and the baseline updates are 99% (figure omitted for space).

As we discuss in §3, layer reuse is low in real-world containers, and even the on-demand "lazy" approach of eStargz must still fetch file metadata from all layers. Conversely, Starlight optimizes updates at a finer file-level granularity, and also stores all file metadata at the beginning of the delta bundle. The result is that Starlight is much better able to exploit redundancy in updates, significantly outperforming the benchmark approaches.

Effect of Bandwidth Can increasing bandwidth help mitigate slow provisioning time? We find that higher bandwidth does not provide a corresponding improvement in provisioning time at higher RTT, even for the baseline approach at 0.15ms. This is not surprising: container provisioning is not purely bandwidth-bound task, since we must also decompress and start containers.

Very low bandwidth We repeated our experiments with a 5Mbps network. At such low bandwidth, transmission time overwhelms the effect of latency: normalized provisioning time for fresh deployments is 9–10.5× higher (compared to 100Mbps network with 0.15ms RTT) for baseline and wget, while eStargz and Starlight reduce it to 2.5–4×. For up-
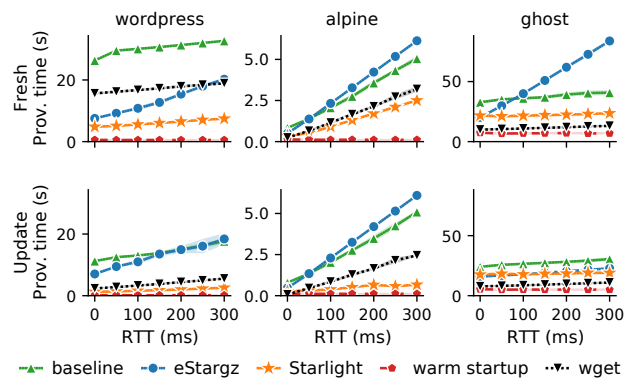
Figure 5: Provisioning times versus round-trip latency for selected containers. Shaded areas show standard deviation.

dates, the baseline is 8–9×, wget and eStargz are 2.5–3×, and Starlight the fastest at 1× across the range of RTT values.

Interestingly, the network is so slow that Flink class loader times out when opening one of the class files when provisioning with eStargz and Starlight. This is the only case we have found of timeout due to on-demand downloading. Indeed, such timeouts are very rare in practice since most software does not timeout on read-only open() calls, and software that does must handle timeouts correctly to function with NFS mounts and other distributed filesystems. Nevertheless, we could mitigate such issues by automatically or manually sorting these files earlier in the delta bundle. Starlight's on-demand optimizer makes this straightforward.
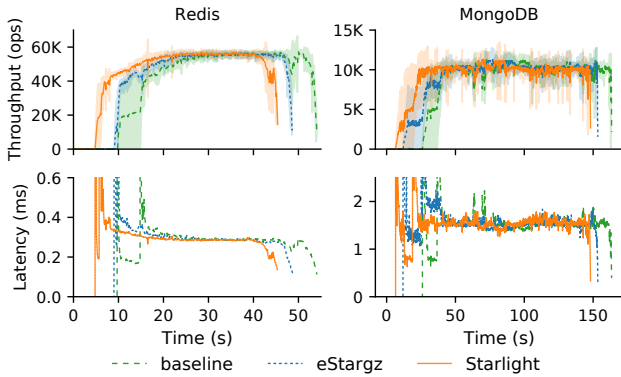
Figure 6: Redis (left) and MongoDB (right) performance during provisioning. Shaded areas show standard deviation.

Figure 7: Scalability of Starlight proxy as the achieved transfer rate for different number of concurrent workers. Network bandwidth is capped at 5Gbps, and RTT is ∼0.15ms.

Individual Analysis Figure 5 shows provisioning time of selected containers across a range of latencies at 100Mbps.

We find that eStargz is bottlenecked by queuing delays caused by on-demand file downloads, and can be slower than even the baseline for RTT over 50ms. Starlight outperforms both except for ghost at 0ms, which is the worst case for Starlight: a 84K file container whose delta bundle takes 3 seconds to build. See Appendix A.2 for in-depth analysis.

5.3 Performance

We measure application, worker, and proxy performance. Unless otherwise noted, proxy-worker RTT is set to 150ms.

Application Performance Ideally, containers deployed using Starlight would exhibit similar application performance as those deployed using the baseline approach, especially during provisioning when Starlight is decompressing files.

To confirm this, we measure application performance for two databases: Redis (in-memory) and MongoDB (disk-based). We run YCSB [9] Workload A (50% read/write ratio) on a separate `m5a.large` instance as the client while we perform a fresh deployment the containerized application, and measure the throughput and read latency of database operations. We repeat each experiment 5 times; each run consists of 2 million database operations, long enough sufficient to finish provisioning and for application performance to stabilize.

Figure 6 depicts throughput and latency over time for both applications. With Starlight, the worker starts handling requests and finishes processing workload earlier than with the other two methods. Additionally, it reaches the same maximum throughput and minimum query latency.

In summary, Starlight workers exhibit no performance overhead compared to the baseline approach and eStargz, and moreover the time gained by early provisioning directly translates to finishing jobs faster.

Worker CPU Usage and Memory We measured the total CPU time used by the snapshotter and containerd daemons during provisioning of containers in Table 2. CPU usage is largely determined by image size, up to 40 seconds of CPU time for the largest image. Median CPU time was 12 seconds for the baseline, 15.1 seconds for eStargz, and 9.8 seconds for Starlight, since it is more effective in removing cross-layer duplicate files. This is consistent with our finding that containerized application exhibit no performance overhead.
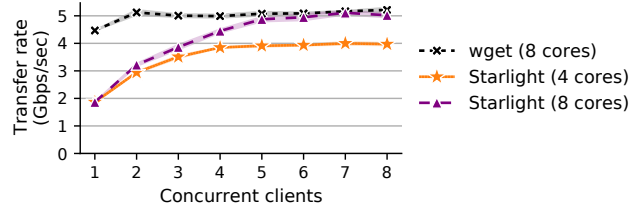
Starlight memory usage, measured as total maximum resident set size of the snapshotter and containerd daemons, is linear in the number of files (140MB plus 9.5KB per file, R^2=0.784) since it maintains file metadata (§4.5 and §4.6). Memory use for both Starlight and eStargz is similar, ranges from under 200MB for most containers to 1GB for ghost – a massive container image with over 84K files. A recent analysis [64] finds that the median container image has 1,090 files, while 70% of images have less fewer 20,000 files – approximately 330MB for Starlight.

Optimization Time Optimizing the delta bundle is by far the most computationally intense operation for the proxy. We find we can compute delta bundles for images of up to 30K files in under one second (figure omitted for space), which includes most of Table 2; the sole exception is ghost at 84K files, which takes three seconds. Similarly, 80% of the container images in Docker Hub [64] have fewer than 30K files, and could therefore be processed within one second. Finally, the time to build delta bundle could be eliminated completely for common deployments by placing a cache in front of the Starlight proxy; we do not do so in any of our experiments.

Scalability We use Apache Benchmark to measure the achievable transfer rate of the Starlight proxy as we increase the number of concurrent clients repeatedly requesting the Redis delta bundle (36.8MB). This is equivalent to the common setup where hundreds of simultaneous Starlight worker requests are load-balanced across multiple replicas of the proxy, and the goal is to saturate the bandwidth – if the proxy is network bound, we are serving as many clients as the network supports. For this experiment, we run with no artificial bandwidth or latency limits. For reference, we request an image of equivalent size from an nginx webserver. Figure 7 shows a Starlight proxy running on a 4-core instance is able to saturate about 80% of the link bandwidth before becoming
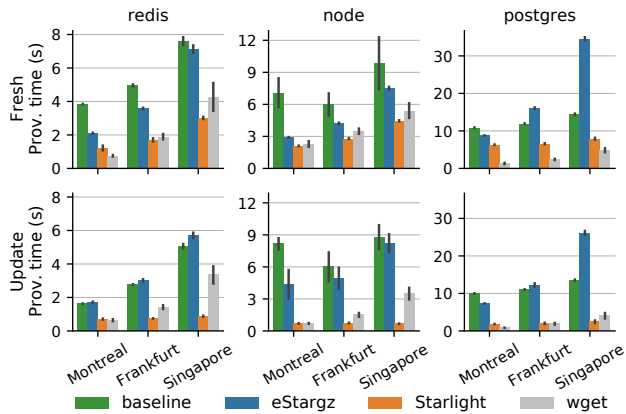
Figure 8: Provisioning time when moving the worker between different datacenters. Errors bars show standard deviation. The registry is located in North Virginia.

bottlenecked due to the need to optimize the delta bundle. Once we switch to 8 cores, it becomes network bound.

5.4 Geo-Distributed WAN Experiment

Thus far we have evaluated Starlight using controlled experiments in a single AWS datacenter. Here, we Starlight performance in a multi-cloud (wide area network) setup running in multiple datacenters over the real network. We place the registry in `us-east-1` region (N. Virginia) and move the worker to increasingly distant locations: `ca-central-1` (average RTT to registry 14ms, bandwidth 4.24Gbps), `eu-central-1` (89ms, 2.79Gbps), and `ap-southeast-1` (209ms, 1.15Gbps).

Figure 8 shows the provisioning time for fresh and update deployment. Results generally match our previous observations: Starlight substantially outperforms the baseline and eStargz, and in many cases is faster than a simple wget of the delta bundle. eStargz is sensitive to increased latency, in some cases becoming slower than the baesline approach. Finally, Starlight support for container updates is much more effective than the other approaches, and can reduce provisioning time to a fraction of the other approaches.

6 Related Work

There are several streams of work on container provisioning.

On-Demand Download Slacker [28] starts containers early and uses NFS to load files on-demand without requiring the entire container image. CRFS [27] follows a similar idea, but uses a seekable tar gzip format with more efficient compression, allowing it to work with standard registries. DADI [37] also uses on-demand fetching but operates at the block level, which requires a customized image format and registry. eStargz [58] uses collected filesystem traces to identify files

needed during provisioning and prefetch them first, before switching to on-demand downloading. Starlight also sorts files based on collected traces, but its push-based design scales better with higher latency. Moreover, Starlight's protocol is file-based rather than layer-based as prior approaches.

Peer-to-peer Some approaches use workers to help provision other workers, Wharf [65] and Shifter [26] propose client-side image sharing: workers act as caches, serving locally stored images to other workers. FID [32], CoMI-Con [44], and Kraken [31] are P2P docker registries that help reduce registry load by utilizing the bandwidth of workers in the datacenter. Similarly, FaaSNet [60] uses a tree of workers to accelerate provisioning inside datacenters for scsaling Function-as-as-Service workloads inside a datacenter. These approaches tend to focus on single datacenter setting with the goal of reducing registry load. They may not be applicable outside the datacenter or where bandwidth and other worker resources are limited. Conversely, Starlight is focused on accelerating provisioning without increasing worker load.

Registry optimizations Fu et al. [25] and Anwar et al. [4] proposes smart caching and prefetching image layers from the back-end object store to the registry using the production workload, in order to do large scale software provisioning. Starlight is orthogonal to, and compatible with, these works since it does not require changing the registry.

7 Conclusion

Containers have evolved in a single datacenter environment, but are increasingly used in geo-distributed settings such as edge, mobile, and multi-cloud environments. We revisit several of the design decisions behind containers, and show that while they are convenient for developers, they slow down provisioning. Starlight redesigns the provisioning pipeline to support faster container deployment, while maintaining the layer-based structure that makes containers easy to develop and maintain. Empirical evaluation using a large set of popular containers shows Starlight provisioning times are significantly smaller than existing approaches, while incurring no performance overhead. Moreover, Starlight is backwards compatible and makes use of existing registries. Starlight is available as an open-source project at: https://github.com/mc256/starlight.

Starlight's design opens several avenues for improvement. For example, since the delta bundle is optimized on-demand, we can improve it and even tailor it to specific scenarios by collecting traces online during deployment, or by training an ML model to predict which files will be needed first. Another improvement is support for repurposing workers: by modifying the optimizer and extending the delta bundle design, we could optimize switching between arbitrary sets of containers.

References

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, Santa Clara, CA, February 2020. USENIX Association.

[2] Werner Almesberger. Linux traffic control - implementation overview. Technical report, EPFL ICA, 1998.

[3] Amazon. Amazon Elastic Container Service (Amazon ECS). https://aws.amazon.com/ecs/.

[4] Ali Anwar, Mohamed Mohamed, Vasily Tarasov, Michael Littley, Lukas Rupprecht, Yue Cheng, Nannan Zhao, Dimitrios Skourtis, Amit S. Warke, Heiko Ludwig, Dean Hildebrand, and Ali R. Butt. Improving docker registry design based on production workload analysis. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 265–278, Oakland, CA, February 2018. USENIX Association.

[5] A. Balalaie, A. Heydarnoori, and P. Jamshidi. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software*, 33(3):42–52, 2016.

[6] Steve Beattie, Seth Arnold, Crispin Cowan, Perry Wagle, Chris Wright, and Adam Shostack. Timing the application of security patches for optimal uptime. In *16th Systems Administration Conference (LISA 02)*, Philadelphia, PA, November 2002. USENIX Association.

[7] Containerd. Snapshots design. https://github.com/containerd/containerd/blob/main/design/snapshots.md.

[8] Containerd. Stargz snapshotter. https://github.com/containerd/stargz-snapshotter.

[9] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[10] Lorenzo Corneo, Maximilian Eder, Nitinder Mohan, Aleksandr Zavodovski, Suzan Bayhan, Walter Wong, Per Gunningberg, Jussi Kangasharju, and Jörg Ott. *Surrounded by the Clouds: A Comprehensive Cloud Reachability Study*, page 295–304. Association for Computing Machinery, New York, NY, USA, 2021.

[11] Breno Costa, Joao Bachiega, Leonardo Rebouças de Carvalho, and Aleteia P. F. Araujo. Orchestration in fog computing: A comprehensive survey. *ACM Comput. Surv.*, 55(2), January 2022.

[12] Geoff Cumming, Fiona Fidler, and David L. Vaux. Error bars in experimental biology . *Journal of Cell Biology*, 177(1):7–11, 04 2007.

[13] Richard Cziva and Dimitrios P. Pezaros. Container network functions: Bringing nfv to the network edge. *IEEE Communications Magazine*, 55(6):24–31, 2017.

[14] Bradley Denby and Brandon Lucia. Orbital edge computing: Nanosatellite constellations as a new class of computer system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 939–954, New York, NY, USA, 2020. Association for Computing Machinery.

[15] Docker. Best practices for writing dockerfiles. https://docs.docker.com/develop/develop-images/dockerfile_best-practices/.

[16] Docker. Docker compose. https://github.com/docker/compose.

[17] Docker. Docker documentation. https://docs.docker.com/engine/reference/commandline/dockerd/.

[18] Docker. Empowering app development for developers | docker. https://www.docker.com/.

[19] Philip J. Fleming and John J. Wallace. How not to lie with statistics: The correct way to summarize benchmark results. *Commun. ACM*, 29(3):218–221, March 1986.

[20] The Linux Foundation. Cloud native computing foundation. https://cncf.io.

[21] The Linux Foundation. containerd: An industry-standard container runtime with an emphasis on simplicity, robustness and portability. https://containerd.io/.

[22] The Linux Foundation. K3s: Lightweight kubernetes. https://k3s.io.

[23] The Linux Foundation. Kubernetes. https://kubernetes.io/.

[24] The Linux Foundation. Open container initiative. https://opencontainers.org/.

[25] Silvery Fu, Radhika Mittal, Lei Zhang, and Sylvia Ratnasamy. Fast and efficient container startup at the edge via dependency scheduling. In *3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20)*. USENIX Association, June 2020.

[26] Lisa Gerhardt, Wahid Bhimji, Shane Canon, Markus Fasel, Doug Jacobsen, Mustafa Mustafa, Jeff Porter, and Vakho Tsulaia. Shifter: Containers for HPC. In *Journal of physics: Conference series*, volume 898, page 082021. IOP Publishing, 2017.

[27] Google. CRFS: Container registry filesystem. https://github.com/google/crfs.

[28] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Slacker: Fast distribution with lazy docker containers. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 181–195, Santa Clara, CA, February 2016. USENIX Association.

[29] Stephen Hemminger. Network emulation with NetEm. In *Linux Conf Australia*, pages 18–23, 2005.

[30] Docker Inc. Docker Hub: Container image library | app containerization. https://registry.hub.docker.com/.

[31] Uber Inc. Kraken - p2p docker registry capable of distributing tbs of data in seconds. https://github.com/uber/kraken.

[32] Wang Kangjin, Yang Yong, Li Ying, Luo Hanmei, and Ma Lin. Fid: A faster image distribution system for docker platform. In *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, pages 191–198, 2017.

[33] The kernel development community. Fuse the linux kernel documentation. https://www.kernel.org/doc/html/latest/filesystems/fuse.html.

[34] kernel.org. Overlay filesystem – the linux kernel documentation. https://www.kernel.org/doc/html/latest/filesystems/overlayfs.html.

[35] Petros Koutoupis. Everything you need to know about Linux containers, part i: Linux control groups and process isolation. *Linux Journal*, 2018, 2018.

[36] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. The quic transport protocol: Design and internet-scale deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 183–196, New York, NY, USA, 2017. Association for Computing Machinery.

[37] Huiba Li, Yifan Yuan, Rui Du, Kai Ma, Lanzheng Liu, and Windsor Hsu. DADI: Block-level image service for agile and elastic application deployment. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 727–740. USENIX Association, July 2020.

[38] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 218–233, New York, NY, USA, 2017. Association for Computing Machinery.

[39] Scott McCarty. A practical introduction to container terminology, 2018. https://developers.redhat.com/blog/2018/02/22/container-terminology-practical-introduction.

[40] Isla Mcketta. How Starlink's satellite internet stacks up against HughesNet and Viasat around the globe, 2021. https://www.speedtest.net/insights/blog/starlink-hughesnet-viasat-performance-q2-2021/.

[41] Microsoft. Azure container instances. https://azure.microsoft.com/en-us/services/container-instances/.

[42] Seyed Hossein Mortazavi, Mohammad Salehe, Moshe Gabel, and Eyal de Lara. Feather: Hierarchical querying for the edge. In *2020 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 271–284, 2020.

[43] Seyed Hossein Mortazavi, Mohammad Salehe, Carolina Simoes Gomes, Caleb Phillips, and Eyal de Lara. Cloudpath: A multi-tier cloud computing framework. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, SEC '17, New York, NY, USA, 2017. Association for Computing Machinery.

[44] Senthil Nathan, Rahul Ghosh, Tridib Mukherjee, and Krishnaprasad Narayanan. CoMICon: A co-operative management system for docker container images. In *2017 IEEE International Conference on Cloud Engineering (IC2E)*, pages 116–126, 2017.

[45] Mahesh Nayak, Kumud Dwivedi, and Cheryl McGuire. Azure network round-trip latency statistics, 2021. https://docs.microsoft.com/en-us/azure/networking/azure-network-latency.

[46] The Containers Organization. Buildah: a tool that facilitates building open container initiative (oci) container images. https://buildah.io/.

[47] Misun Park, Ketan Bhardwaj, and Ada Gavrilovska. Toward lighter containers for the edge. In *3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20)*. USENIX Association, June 2020.

[48] Valerio Persico, Alessio Botta, Pietro Marchetta, Antonio Montieri, and Antonio Pescap. On the performance of the wide-area networks interconnecting public-cloud datacenters around the globe. *Comput. Netw.*, 112(C):67–83, January 2017.

[49] CNCF Distribution Project. Distribution - the toolkit to pack, ship, store, and deliver container content. https://github.com/distribution/distribution.

[50] Prashanth Rajivan, Efrat Aharonov-Majar, and Cleotilde Gonzalez. Update now or later? effects of experience, cost, and risk preference on update decisions. *Journal of Cybersecurity*, 6(1):tyaa002, 2020.

[51] Brian Ramprasad, Alexandre da Silva Veith, Moshe Gabel, and Eyal de Lara. Sustainable computing on the edge: A system dynamics perspective. In *Proceedings of the 22nd International Workshop on Mobile Computing Systems and Applications*, HotMobile '21, page 64–70, New York, NY, USA, 2021. Association for Computing Machinery.

[52] Mahadev Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, 2017.

[53] J. Shah and D. Dubaria. Building modern clouds: Using Docker, Kubernetes & Google Cloud Platform. In *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 0184–0189, 2019.

[54] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[55] Dimitris Skourtis, Lukas Rupprecht, Vasily Tarasov, and Nimrod Megiddo. Carving perfect layers out of docker images. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association.

[56] Jörg Thalheim, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. Cntr: Lightweight OS containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 199–212, Boston, MA, July 2018. USENIX Association.

[57] Abhishek Tiwari, Brian Ramprasad, Seyed Hossein Mortazavi, Moshe Gabel, and Eyal de Lara. Reconfigurable streaming for the mobile edge. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*, HotMobile '19, page 153–158, New York, NY, USA, 2019. Association for Computing Machinery.

[58] Kohei Tokunaga. Startup containers in lightning speed with lazy image distribution on containerd, Apr 2020.

[59] B. Varghese, E. De Lara, A. Ding, C. Hong, F. Bonomi, S. Dustdar, P. Harvey, P. Hewkin, W. Shi, M. Thiele, and P. Willis. Revisiting the arguments for edge computing research. *IEEE Internet Computing*, (01):1–1, jun 5555.

[60] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. FaaS-Net: Scalable and fast provisioning of custom serverless container runtimes at Alibaba cloud function compute. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 443–457. USENIX Association, July 2021.

[61] Ying Xiong, Yulin Sun, Li Xing, and Ying Huang. Extend cloud to edge with KubeEdge. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 373–377, 2018.

[62] Ashkan Yousefpour, Caleb Fung, Tam Nguyen, Krishna Kadiyala, Fatemeh Jalali, Amirreza Niakanlahiji, Jian Kong, and Jason P. Jue. All one needs to know about fog computing and related edge computing paradigms: A complete survey. *Journal of Systems Architecture*, 98:289–330, 2019.

[63] N. Zhao, V. Tarasov, A. Anwar, L. Rupprecht, D. Skourtis, A. Warke, M. Mohamed, and A. Butt. Slimmer: Weight loss secrets for Docker registries. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 517–519, 2019.

[64] Nannan Zhao, Vasily Tarasov, Hadeel Albahar, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Amit S. Warke, Mohamed Mohamed, and Ali R. Butt. Large-scale analysis of the docker hub dataset. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–10, Sep. 2019.

[65] Chao Zheng, Lukas Rupprecht, Vasily Tarasov, Douglas Thain, Mohamed Mohamed, Dimitrios Skourtis, Amit S. Warke, and Dean Hildebrand. Wharf: Sharing docker images in a distributed file system. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 174–185, New York, NY, USA, 2018. Association for Computing Machinery.

A Appendix

A.1 Container Images Used in Evaluation

Table 2 below lists containers and version tags used in our experiments; combined, they have over 15 billion downloads in Docker Hub.

Category	Images
Linux	alpine:3.13.4
	ubuntu:focal-20210401
Web	memcached:1.6.8
	nginx:1.19.10
	httpd:2.4.43
Data	mysql:8.0.23
	mariadb:10.5.8
	redis:6.2.1
	mongo:4.0.23
	postgres:13.1
	rabbitmq:3.8.13
Services	registry:2.7.0
	wordpress:php7.3-fpm
	ghost:3.42.5-alpine
Dev	node:16-alpine3.11
	openjdk:11.0.11-9-jdk
	golang:1.16.2
	python:3.9.3
Edge	flink:1.12.3-scala_2.11-java8
	cassandra:3.11.9
	eclipse-mosquitto:2.0.9-openssl

Table 2: Container images used in our evaluation.

A.2 Analysis of Selected Containers

Figure 5 shows provisioning time of selected containers across a range of latencies at 100Mbps.

When updating wordpress, the baseline approach is able to reuse 4 out of 18 layers, making it faster in update. eStargz, though faster than the baseline approach in fresh deployments, does not benefit much from this layer reuse since it is bottlenecked by on-demand file downloads. Starlight, on the other hand, is much faster than either approach, reducing update provisioning time by approximately $8\times$.

For alpine eStargz is slower than the baseline when RTT is above 50ms. This is because the alpine image is small and its file access pattern is not entirely deterministic. Provisioning time is thus dominated by queuing delays due layer downloads and on-demand file downloads. Starlight also suffers somewhat from out-of-order file accesses, but is still able to deploy the container quickly, and is even faster than wget.

Finally, we discuss ghost – the worst case for Starlight. Starlight's provisioning time with low RTT is 10% higher
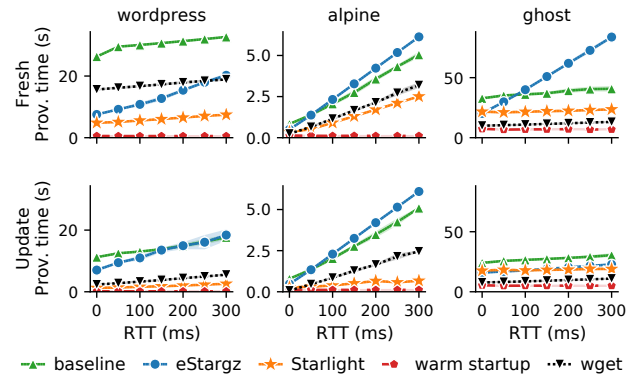
Figure 5: Provisioning times versus round-trip latency for selected containers. Shaded areas show standard deviation. (figure repeated from page 10)

than eStargz's – the only container where this happens. Building a delta bundle takes 3 seconds for this 84K file container. eStargz provisioning time grows quickly with latency, however, and Starlight outperforms it when RTT is above 50ms.

POWERTCP: Pushing the Performance Limits of Datacenter Networks[*]

Vamsi Addanki
TU Berlin
University of Vienna

Oliver Michel
Princeton University
University of Vienna

Stefan Schmid
TU Berlin
University of Vienna

Abstract

Increasingly stringent throughput and latency requirements in datacenter networks demand fast and accurate congestion control. We observe that the reaction time and accuracy of existing datacenter congestion control schemes are inherently limited. They either rely only on explicit feedback about the network state (e.g., queue lengths in DCTCP) or only on variations of state (e.g., RTT gradient in TIMELY). To overcome these limitations, we propose a novel congestion control algorithm, POWERTCP, which achieves much more fine-grained congestion control by adapting to the bandwidth-window product (henceforth called power). POWERTCP leverages in-band network telemetry to react to changes in the network instantaneously without loss of throughput and while keeping queues short. Due to its fast reaction time, our algorithm is particularly well-suited for dynamic network environments and bursty traffic patterns. We show analytically and empirically that POWERTCP can significantly outperform the state-of-the-art in both traditional datacenter topologies and emerging reconfigurable datacenters where frequent bandwidth changes make congestion control challenging. In traditional datacenter networks, POWERTCP reduces tail flow completion times of short flows by 80% compared to DCQCN and TIMELY, and by 33% compared to HPCC even at 60% network load. In reconfigurable datacenters, POWERTCP achieves 85% circuit utilization without incurring additional latency and cuts tail latency by at least 2x compared to existing approaches.

1 Introduction

The performance of more and more cloud-based applications critically depends on the underlying network, requiring datacenter networks (DCNs) to provide extremely low latency and high bandwidth. For example, in distributed machine learning applications that periodically require large data transfers, the network is increasingly becoming a bottleneck [36]. Similarly, stringent performance requirements are introduced by today's trend of resource disaggregation in datacenters where fast access to remote resources (e.g., GPUs or memory) is pivotal

for the overall system performance [36]. Building systems with strict performance requirements is especially challenging under bursty traffic patterns as they are commonly observed in datacenter networks [12, 16, 47, 53, 55].

These requirements introduce the need for fast and accurate network resource management algorithms that optimally utilize the available bandwidth while minimizing packet latencies and flow completion times. Congestion control (CC) plays an important role in this context being ''a key enabler (or limiter) of system performance in the datacenter'' [34]. In fact, fast reacting congestion control is not only essential to efficiently adapt to bursty traffic [29, 48], but is also becoming increasingly important in the context of emerging reconfigurable datacenter networks (RDCNs) [13, 14, 20, 33, 38, 39, 50]. In these networks, a congestion control algorithm must be able to quickly ramp up its sending rate when high-bandwidth circuits become available [43].

Traditional congestion control in datacenters revolves around a bottleneck link model: the control action is related to the state i.e., queue length at the bottleneck link. A common goal is to efficiently control queue buildup while achieving high throughput. Existing algorithms can be broadly classified into two types based on the feedback that they react to. In the following, we will use an analogy to electrical circuits[1] to describe these two types. The first category of algorithms react to the absolute network state, such as the queue length or the RTT: a function of network ''effort'' or **voltage** defined as the sum of the bandwidth-delay product and in-network queuing. The second category of algorithms rather react to variations, such as the change of RTT. Since these changes are related to the network ''flow'', we say that these approaches depend on the **current** defined as the total transmission rate. We tabulate our analogy and corresponding network quantities in Table 1. According to this classification, we call congestion control protocols such as CUBIC [21], DCTCP [7], or Vegas [15] **voltage-based CC** algorithms as

---

[1]This analogy is inspired from S. Keshav's lecture series based on mathematical foundations of computer networking [31]. We emphasize that our power analogy is meant for the networking context considered in this paper and it should not be applied to other domains of science.

Quantity	Analogy
Total transmission rate (network flow)	Current (λ)
BDP + buffered bytes (network effort)	Voltage (v)
Current × Voltage	Power (Γ)

Table 1: Analogy between metrics in networks and in electrical circuits. Note that the network here is the ''pipe'' seen by a flow and not the whole network.
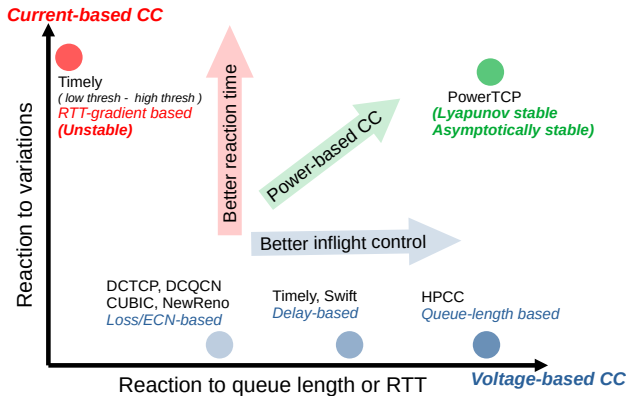
Figure 1: Existing congestion control algorithms are fundamentally limited to a single dimension in their window (or rate) update decisions and are unable to distinguish between two scenarios across multiple dimensions.

they react to absolute properties such as the bottleneck queue length, delay, Explicit Congestion Notification (ECN), or loss. Recent proposals such as TIMELY [41] are **current-based CC** algorithms as they react to the variations, such as the RTT-gradient. In conclusion, we find that existing congestion control algorithms are fundamentally limited to one of the two dimensions (voltage or current) in the way they update the congestion window.

We argue that the input to a congestion control algorithm should rather be a function of the two-dimensional state of the network (i.e., both voltage and current) to allow for more informed and accurate reaction, improving performance and stability. In our work, we show that there exists an accurate relationship between the optimal adjustment of the congestion window, the network voltage and the network current. We analytically show that the optimal window adjustment depends on the product of network voltage and network current. We call this product **network power**: current × voltage, a function of both queue lengths and queue dynamics.

Figure 1 illustrates our classification. Existing protocols depend on a single dimension, voltage or current. This can result in imprecise congestion control as the protocol is unable to distinguish between fundamentally different scenarios, and, as a result, either reacts too slowly or overreacts, both impeding performance. Accounting for both voltage and current, i.e., power, balances accurate inflight control and fast reaction, effectively providing the best of both worlds.

In this paper we present POWERTCP, a novel *power*-based congestion control algorithm that accurately captures both *voltage* and *current* dimensions for every control action using measurements taken within the network and propagated through in-band network telemetry (INT). POWERTCP is able to utilize available bandwidth within one or two RTTs while being stable, maintaining low queue lengths, and resolving congestion rapidly. Furthermore, we show that POWERTCP is Lyapunov-stable, as well as asymptotically stable and has a convergence time as low as five update intervals (Appendix A). This makes POWERTCP highly suitable for today's datacenter networks and dynamic network environments such as in reconfigurable datacenters.

POWERTCP leverages in-network measurements at programmable switches to accurately obtain the bottleneck link state. Our switch component is lightweight and the required INT header fields are standard in the literature [36]. We also discuss an approximation of POWERTCP for use with non-programmable, legacy switches.

To evaluate POWERTCP, we focus on a deployment scenario in the context of RDMA networks where the CC algorithm is implemented on a NIC. Our results from large-scale simulations show that POWERTCP reduces the 99.9-percentile short flow completion times by 80% compared to DCQCN [56] and by 33% compared to the state-of-the-art low-latency protocol HPCC [36]. We show that POWERTCP maintains near-zero queue lengths without affecting throughput or incurring long flow completion times even at 80% load. As a case study, we explore the benefits of POWERTCP in reconfigurable datacenter networks where it achieves 80 − 85% circuit utilization and reduces tail latency by at least 2× compared to the state-of-the-art [43]. Finally, as a proof-of-concept, we implemented POWERTCP in the Linux kernel and the telemetry component on an Intel Tofino programmable line-rate switch using P4 [18].

In summary, our key contributions in this paper are:

- We reveal the shortcomings of existing congestion control approaches which either only react to the current state or the dynamics of the network, and introduce the notion of *power* to account for both.

- POWERTCP, a power-based approach to congestion control at the end-host which reacts faster to changes in the network such as an arrival of burst, fluctuations in available bandwidth etc.,

- An evaluation of the benefits of POWERTCP in traditional DCNs and RDCNs.

- As a contribution to the research community and to facilitate future work, all our artefacts have been made publicly available at:
  https://powertcp.self-adjusting.net.
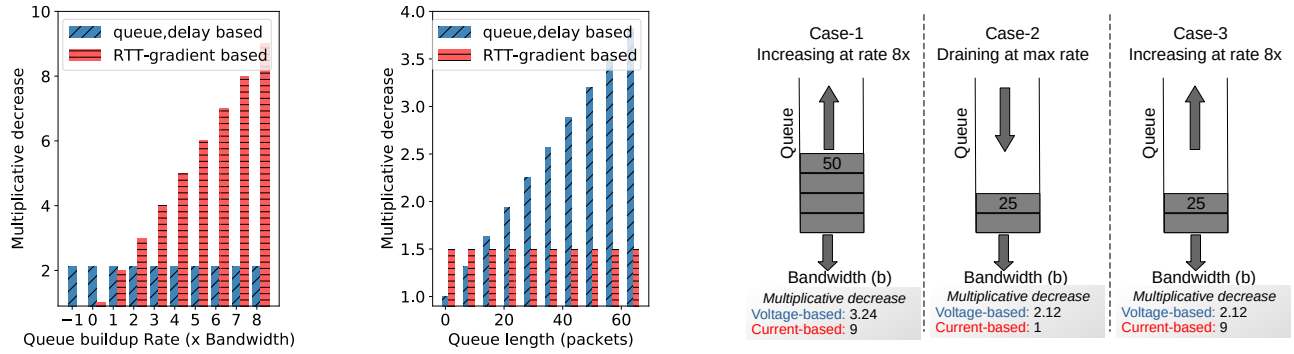
(a) Voltage-based CC is oblivious to queue buildup rate.

(b) Current-based CC is oblivious to queue lengths.

(c) Voltage-based CC cannot differentiate case-2 vs case-3; whereas current-based CC cannot differentiate case-1 vs case-3.

Figure 2: Existing CC schemes, classified as voltage and current-based, are orthogonal in their response to queue length and queue buildup rate.

2 Motivation

We first provide a more detailed motivation of our work by highlighting the benefits and drawbacks of existing congestion control approaches. In the following, **voltage-based CC** refers to the class of end-host congestion control algorithms that react to the state of the network in absolute values related to the bandwidth-delay product, such as bottleneck queue length, delay, loss, or ECN; **current-based CC** refers to the class of algorithms that react to changes in the state, such as the RTT-gradient. Voltage-based CC algorithms are likely to exhibit better stability but are fundamentally limited in their reaction time. Current-based CC algorithms detect congestion faster but ensuring stability may be more challenging. Indeed, TIMELY [41], a current-based CC, deployed at Google datacenters, turned out to be unstable [57] and evolved to SWIFT [34], a voltage-based CC.

Orthogonal to our approach, receiver-driven transport protocols [22, 26, 42] have been proposed which show significant performance improvements. A receiver-driven transport approach relies on the assumption that datacenter networks are well-provisioned and claims that congestion control is unnecessary; for example ''NDP performs no congestion control whatsoever in a Clos topology'' [22]. The key difference is that receiver-driven approaches take feedback from the ToR downlink at the receiver which can only identify congestion at the last hop, whereas sender-based approaches rely on a variety of feedback signals to identify congestion anywhere along the path. In this paper, we focus on the sender-based congestion control approach which can in principle handle congestion anywhere along the round-trip path between a sender and a receiver, even in oversubscribed datacenters.

To take a leap forward and design fine-grained datacenter congestion control algorithms, we present an analytical approach and study the fundamental problems faced by existing algorithms. We first formally express the desirable properties of a datacenter congestion control law (§2.1) and then analytically identify the drawbacks of existing control laws

(§2.2). Finally, we discuss the lessons learned and formulate our design goals (§2.3).

2.1 Desirable Control Law Properties

Among various desired properties of datacenter congestion control, high throughput and low tail latency are most important [7, 36, 41] with fairness and stability being essential as well [54, 57]. Achieving these properties simultaneously can be challenging. For example, to realize high throughput, we may aim to keep the queue length at the bottleneck link large; however, this may increase latency. Thus, an ideal CC algorithm must be capable of maintaining near-zero queue lengths, achieving both high throughput and low latency. It must further minimize throughput loss and latency penalty caused by perturbations, such as bursty traffic.

In order to formalize our requirements, we consider a single-bottleneck link model widely used in the literature [24, 40, 54, 57]. Specifically, we assume that all senders use the same protocol, transmit long flows[2] sharing a common bottleneck link with bandwidth b, and have a base round trip time τ (excluding queuing delays). In this model, equilibrium is a state reached when the window size and bottleneck queue length stabilize. We now formally express the desired equilibrium state that captures our performance requirements in terms of the sum of window sizes of all flows (aggregate window size) $w(t)$, bandwidth delay product $b \cdot \tau$, and bottleneck queue length $q(t)$:

$$0 < q(t) < \varepsilon \qquad (1)$$
$$b \cdot \tau \leq w(t) < b \cdot \tau + \varepsilon$$
$$\dot{q}(t) = 0; \ \dot{w}(t) = 0$$

where ε is a positive integer. First, this captures the requirement for high throughput i.e., when $w(t) > b \cdot \tau$ and $q(t) > 0$, the number of inflight bytes are greater than the bandwidth-delay product (BDP) and the queue length is greater than zero.

---

[2]Note that, although most DC flows are short flows, most DC traffic volume (bytes) is from long flows [7, 9].
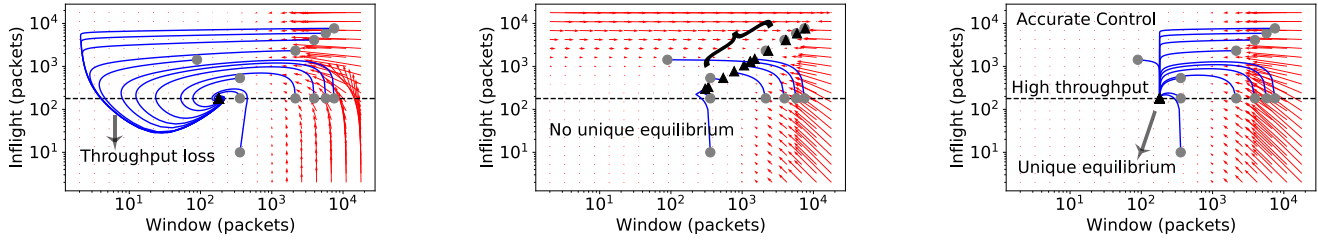
(a) Voltage-based CC (RTT or queue length) exhibits equilibrium properties but has an imprecise reaction leading to throughput loss.

(b) Current-based CC (RTT-gradient) reacts faster but has no unique equilibrium point, and is thereby unable to stabilize queue lengths.

(c) POWERTCP, a power-based CC, exhibits equilibrium properties and has a precise reaction to perturbations.

Figure 3: Phase plots showing the trajectories of existing schemes and our approach POWERTCP from different initial states (circles) to equilibrium (triangles). At each point on the plane, arrows show the direction in which the system moves. An example is depicted with bottleneck link bandwidth 100Gbps and a base RTT of $20\mu s$. BDP is shown by a horizontal dotted line and any trajectory going below this line indicates throughput loss.

Second, from $w(t) < b \cdot \tau + \varepsilon$ and $q(t) < \varepsilon$, the queue length is at most ε, thereby achieving low latency. Finally, for the system to stabilize, we need that $\dot{q}(t) = 0$ and $\dot{w}(t) = 0$.

As simple as these requirements are, it is challenging to control the aggregate window size $w(t)$ while CC operates per flow. In addition to the equilibrium state requirement, we need fast response to perturbations. The response must minimize the distance from the equilibrium i.e., minimize the latency or throughput penalty caused by a perturbation (e.g., incast or changes in available bandwidth).

In this work, we ask two fundamental questions:

(Q1) **Equilibrium point:** Do existing algorithms satisfy the equilibrium state in Eq. 1 for the aggregate window size? In addition to the equilibrium behavior, we are also interested in the reaction to a perturbation.

(Q2) **Response to perturbation:** What is the trajectory followed after a perturbation, i.e., the dynamics of the bottleneck queue as well as the TCP window sizes, from an initial point to the equilibrium point?

2.2 Drawbacks of Existing Control Laws

We now aim to analytically answer our questions above and shed light on the inefficiencies of existing protocols, both voltage-based and current-based. We begin by simplifying the congestion avoidance model of existing CC approaches we are interested in, specifically delay, queue length, and RTT-gradient based CC approaches as follows:

$$w_i(t + \delta t) = \gamma \cdot \left(w_i(t) \cdot \frac{e}{f(t)} + \beta \right) + (1 - \gamma) \cdot w_i(t) \quad (2)$$

Here w_i is the window of a flow i, β is the additive increase term, e is the equilibrium point that the algorithm is expected to reach, $f(t)$ is the measured feedback and γ is the exponential moving average parameter. A queue length-based CC [36] sets the desired equilibrium point e as $b \cdot \tau$ (BDP) and the feedback $f(t)$ as the sum of bottleneck queue length and BDP i.e.,

voltage (v). A delay-based CC [34] sets e to τ (base RTT) and the feedback $f(t)$ as RTT which is the sum of queuing delay and base RTT i.e., $\frac{\text{voltage}}{\text{bandwidth}}$ $\left(\frac{v}{b} \right)$. Similarly, the RTT-gradient approach [41] sets e to 1 and the feedback $f(t)$ as one plus RTT-gradient i.e., $\frac{\text{current}}{\text{bandwidth}}$ $\left(\frac{\lambda}{b} \right)$. In Appendix B, we further justify how Eq. 2 captures existing control laws[3]. Note that our simplified model does not capture loss/ECN-based CC algorithms; however, there exists rich literature on the analysis of loss/ECN-based CC algorithms [24, 37] including DCTCP [7, 8]. We now use Euler's first order approximation to obtain the window dynamics as follows:

$$\dot{w}_i(t) = \frac{\gamma}{\delta t} \cdot \left(w_i(t) \cdot \frac{e}{f(t)} - w_i(t) + \beta \right) \quad (3)$$

Each flow i has a sending rate λ_i and hence the bottleneck queue experiences an aggregate arrival rate of λ. In our analogy, λ is the network current. We additionally use the traditional model of queue length dynamics which is independent of the control law [24, 40]:

$$\dot{q}(t) = \begin{cases} \lambda(t - t^f) - \mu(t) & q(t) > 0 \\ 0 & otherwise \end{cases} \quad (4)$$

where $\lambda(t) = \frac{w(t)}{\theta(t)}$. An equilibrium point is the window size w_e and queue length q_e that satisfies $\dot{w}(t) = 0$ and $\dot{q}(t) = 0$.

We are now ready to answer the questions raised.

Equilibrium point: It is well-known from literature that loss/ECN-based schemes operate by maintaining a standing queue [8, 24, 27]. For example, TCP NewReno flows fill the queue to maximum (say q_{max}) and then react by reducing windows by half. Consequently, the bottleneck queue-length oscillates between q_{max} and $q_{max} - b \cdot \tau$ or zero if $q_{max} < b \cdot \tau$. DCTCP flows oscillate around the marking threshold $K > \frac{b \cdot \tau}{7}$

---

[3]TIMELY, for example, is rate-based while our simplification is window-based. However, window and rate are interchangeable for update calculations.

which depends on BDP [7]. This does not satisfy our stringent requirement in Eq. 1. While ECN-based schemes reduce the amount of standing queue required, we still consider the standing queue which is proportional to bandwidth to be unacceptable given the increasing gap between bandwidth vs switch buffers.

It can be shown that there exists a unique equilibrium point for queue length and delay approaches (voltage-based CC) defined by Eq. 2. However, current-based CC and, in particular, RTT-gradient approaches do not have a unique equilibrium point suggesting a lack of control over queue lengths. Intuitively, RTT-gradient approaches quickly adapt the sending rate to stabilize the RTT-gradient ($\dot{\theta} = \frac{\dot{q}}{b}$) which in turn only stabilizes the queue length gradient $\dot{q}(t)$ but fails to control the absolute value of the queue length. It has indeed been shown that TIMELY, a current-based CC does not have a unique equilibrium [57].

Figure 3 visualizes the system behavior according to the window dynamics in Eq. 3 and the queue dynamics in Eq. 4. In Figure 3a we can see that voltage-based CC eventually reaches a unique equilibrium point. In contrast, in Figure 3b we see that current-based CC reaches different final points for different initial points, indicating that there exists no unique equilibrium point thereby violating the desired equilibrium state properties (Eq. 1). To give more context on this observation, in Figure 2 we show the reactions of different schemes for observed queue lengths and queue buildup rate. In Figure 2b, we can see that current-based CC has the same reaction for different queue lengths but exhibits a proportional reaction to queue buildup rate (Figure 2a); consequently, current-based CC cannot stabilize at a unique equilibrium point. Due to space constraints, we move the detailed proof of equilibrium points to Appendix B.

Takeaway. While voltage-based CC can in principle meet the desired equilibrium state requirements in Eq. 1, current-based CC cannot.

Response to perturbation: We observe an orthogonal behavior in the responses of voltage-based CC and current-based CC. In Figure 2b we show that voltage-based CC has a proportional reaction to increased queue lengths but a current-based CC approach has the same response for any queue length. Further in Figure 2a we observe that current-based CC has a proportional reaction to the rate at which queue is building up but a voltage-based CC has the same reaction for any rate of queue build up. This orthogonality in existing schemes often results in scenarios with either insufficient reaction or overreaction. To underline our observation, we use the system of differential equations (Eq. 3 and Eq. 4) to observe the trajectories taken by different control laws after a perturbation. We show the trajectories in Figure 3. Specifically, Figure 3a shows that voltage-based CC (queue length or delay based) eventually reaches a unique equilibrium point but overreacts in the response and losing throughput (window < BDP and $q(t) = 0$) almost for every initial point. In Figure 3b we observe that current-based CC (RTT-gradient) reaches different

end points for different initial states and consequently does not have a single equilibrium point. However, we see that the initial response is faster with current-based CC due to their use of RTT-gradient which is arguably a superior signal to detect congestion onset even at low queue lengths.

Takeaway. Current-based CC is superior in terms of fast reaction but lacks equilibrium state properties while voltage-based CC eventually reaches a unique equilibrium but overreacts in its response for almost any initial state resulting in long trajectories from initial state to equilibrium state.

2.3 Lessons Learned and Design Goals

From our analysis we derive two key observations. First, both voltage and current-based CC have individual benefits. Particularly, voltage-based CC is desirable for the stringent equilibrium properties we require and current-based CC is desirable for fast reaction. Second, both voltage and current-based CC have drawbacks. On one hand, voltage-based CC is oblivious to congestion onset at low queue lengths and on the other hand current-based CC is oblivious to the absolute value of queue lengths. Moreover, voltage-based CC overreacts when the queue drains essentially losing throughput immediately after.

Based on these observations, our goal is to design a control law that systematically combines both voltage and current for every window update action. Specifically our aim is to design a congestion control algorithm with *(i)* equilibrium properties from Eq. 1 exhibited by voltage-based CC and *(ii)* fast response to perturbation exhibited by current-based CC. The challenges are to avoid inheriting the drawbacks of both types of CC, stability and fairness. However in order to design such a control law we face the following challenges:

- Finding an accurate relationship between window, voltage and current. ▷ Property 1

- Ensuring stability, convergence and fairness. ▷ Theorem 1, 2, 3

3 Power-Based Congestion Control

Reflecting on our observations in §2, we seek to design a congestion control algorithm that systematically reacts to both the absolute value of the bottleneck queue length and its rate of change. Our aim is to address today's datacenter performance requirements in terms of high throughput, low latency, and fast reaction to bursts and bandwidth fluctuations.

3.1 The Notion of Power

To address the challenges faced by prior datacenter congestion control algorithms and to optimize along both dimensions, we introduce the notion of *power* associated with the network pipe. Following the bottleneck link model from literature [24, 40], from Eq. 4 we observe that the window size is indeed related to the product of network voltage and network current which we call *power* (Table 1). This corresponds to the product of *(i)* total sending rate λ (current) and *(ii)* the

sum of BDP plus the accumulated bytes q at the bottleneck link (voltage), formally expressed in Eq. 5.

$$\underbrace{\Gamma(t)}_{\text{power}} = \underbrace{(q(t) + b \cdot \tau)}_{\text{voltage}} \cdot \underbrace{\lambda(t - t^f)}_{\text{current}} \quad (5)$$

Notice that the unit of power is $\frac{bit^2}{second}$. We will show the useful properties of power specifically under congestion. Using Eq. 4, we can rewrite Eq. 5 in terms of queue length gradient $\dot{q}$ and the transmission rate μ as,

$$\Gamma(t) = (q(t) + b \cdot \tau) \cdot (\dot{q}(t) + \mu(t)) \quad (6)$$

We now derive a useful property of power using Eq. 6 and Eq. 4 showing an accurate relationship of power and window.

Property 1 (Relationship of Power and Congestion Window)**.**
Power is the bandwidth-window product

$$\Gamma(t) = b \cdot w(t - t^f)$$

Note that the property is over the aggregate window size i.e., the sum of window sizes of all flows sharing the common bottleneck. We emphasize that our notion of power is intended for the networking context and cannot be applied to other domains of science. In the following, we outline the benefits of considering the notion of power and how Property 1 can be useful in the context of congestion control.

3.2 Benefits of Power-Based CC

A power-based control law can exploit Property 1 to precisely update per flow window sizes. Accurately controlling aggregate window size is a key challenge for an end-host congestion control algorithm. A power-based CC overcomes this challenge by gaining precise knowledge about the aggregate window size from measured power. First, using power enables the window update action to account for the bottleneck queue lengths as well as the queue build-up rate. As a result, a power-based CC can rapidly detect congestion onset even at very low queue lengths. At the same time, a power-based CC also reacts to the absolute value of queue lengths, effectively dampening perturbations. Second, calculating power at the end-host requires no extra measurement and feedback mechanisms compared to INT based schemes such as HPCC [36].

3.3 The POWERTCP Algorithm

Driven by our observations, we carefully designed our control law based on power, capturing a systematic reaction to voltage (related to bottleneck queue length), as well as to current (related to variations in the bottleneck queue length).

Control law: POWERTCP is a window-based congestion control algorithm and updates its window size upon receipt of an acknowledgment. For a flow i, every window update is based on *(i)* current window size $w_i(t)$, *(ii)* additive increase β, *(iii)* window size at the time of transmission of the acknowledged segment $w_i(t - \theta(t))$, and *(iv)* power measured from

the feedback information. We refer the reader to Table 2 for the general notations being used. Formally, POWERTCP's control law can be expressed as

$$w_i(t) \leftarrow \gamma \cdot \left(w_i(t - \theta(t)) \cdot \frac{e}{f(t)} + \beta \right) + (1 - \gamma) \cdot w_i(t) \quad (7)$$

$$e = b^2 \cdot \tau; \quad f(t) = \Gamma(t - \theta(t) + t^f)$$

where $\gamma \in (0, 1]$ and β are parameters to the control law. The base round trip time τ must be configured at compile time. If baseRTT is not precisely known, an alternative is to keep track of minimum observed RTT. We first describe how power Γ is computed and then present the pseudocode of POWERTCP in Algorithm 1.

Feedback: POWERTCP's control law is based on power. Note that power (Eq. 5) is only related to variables at the bottleneck link. In order to measure power, we leverage in-band network telemetry. Specifically, the workings of INT and the header fields required are the same as in HPCC (Figure. 4 in [36]). When a TCP sender sends out a packet `P` into the network, it additionally inserts an INT header `INT` into the packet. Each switch along the path then pushes metadata containing the egress queue length (*qlen*), timestamp (*ts*), so far transmitted bytes (*txBytes*), and bandwidth (*b*). All values correspond to the time when the packet is scheduled for transmission. At the receiver, the received packet `P INT 1,2,...r` is read and the INT information is copied to the acknowledgment *ACK* packet `A INT 1,2,...r`. The sender then receives an *ACK* with an INT header and metadata inserted by all the switches along the path from sender to receiver and back to sender `A INT 1,2,...r INT ....n`. Here, the INT header and meta-data pushed by switches along the path serve as feedback and as an input to the CC algorithm.

Accounting for the old window sizes: POWERTCP's control law (Eq. 7) uses the past window size in addition to the current window size to compute the new window size. POWERTCP accounts for old window size by remembering current window size once per RTT.

Algorithm: Putting it all together, we now present the workflow of POWERTCP in Algorithm 1. Upon the receipt of a new acknowledgment (line 2), POWERTCP: *(i)* retrieves the old *cwnd* (line 3), *(ii)* computes the normalized power (line 19) i.e., $\frac{f(t)}{e}$ in Eq. 7, *(iii)* updates *cwnd* (line 5), *(iv)* sets the pacing rate (line 6), and *(v)* remembers the INT header metadata and updates the old *cwnd* once per RTT based on the *ack* sequence number (line 7).

Specifically, power is calculated in the function call to NORMPOWER. First, the gradient of queue lengths is obtained from the difference in queue lengths and difference in timestamps corresponding to an egress port (line 12). Then the transmission rate of the egress port is calculated from the difference in *txBytes* and timestamps (line 13). Current is calculated by adding the queue gradient and transmission rate (line 14). Then, the sum of BDP and the queue length gives

voltage (line 16). Finally, power is calculated by multiplying current and voltage (line 17). We calculate the base power (line 18) and obtain the normalized power (line 19). The normalized power is calculated for each egress port along the path and the maximum value is smoothed and used as an input to the control law.

Finally, the congestion window is updated in the function call to UPDATEWINDOW (line 26) where γ is the exponential moving average parameter and β is the additive increase parameter, both being parameters to the control law (Eq. 7)

Algorithm 1: POWERTCP

```
1  /* ack contains an INT header with
      sequence of per-hop egress port
      meta-data accessed as ack.H[i]      */
```
Input : *ack* and *prevInt*
Output : *cwnd, rate*
2 **procedure** NEWACK(*ack*):
3     $cwnd_{old} = \text{GETCWND}(ack.seq)$
4     $normPower = \text{NORMPOWER}(ack)$
5   UPDATEWINDOW(*normPower, cwnd_{old}*)
6     $rate = \frac{cwnd}{\tau}$
7     $prevInt = ack.H$; UPDATEOLD(*cwnd, ack.seq*)
8 **function** NORMPOWER(*ack*):
9     $\Gamma_{norm} = 0$
10   **for** *each egress port i on the path* **do**
11       $dt = ack.H[i].ts - prevInt[i].ts$
12       $\dot{q} = \frac{ack.H[i].qlen - prevInt[i].qlen}{dt}$ $\triangleright \frac{dq}{dt}$
13       $\mu = \frac{ack.H[i].txBytes - prevInt[i].txBytes}{dt}$ $\triangleright$ txRate
14       $\lambda = \dot{q} + \mu$ $\triangleright \lambda$: Current
15       $BDP = ack.H[i].b \times \tau$
16       $\nu = ack.H[i].qlen + BDP$ $\triangleright \nu$: Voltage
17       $\Gamma' = \lambda \times \nu$ $\triangleright \Gamma'$: Power
18       $e = (ack.H[i].b)^2 \times \tau$
19       $\Gamma'_{norm} = \frac{\Gamma'}{e}$ $\triangleright \Gamma'_{norm}$:Normalized power
20   **if** $\Gamma' > \Gamma_{norm}$ **then**
21         $\Gamma_{norm} = \Gamma'$; $\Delta t = dt$
22   **end if**
23   **end for**
24     $\Gamma_{smooth} = \frac{\Gamma_{smooth} \cdot (\tau - \Delta t) + \Gamma_{norm} \cdot \Delta t}{\tau}$ $\triangleright$ Smoothing
25   **return** Γ_{smooth}
26 **function** UPDATEWINDOW(*power, ack*):
27     $cwnd = \gamma \times (\frac{cwnd_{old}}{normPower} + \beta) + (1 - \gamma) \times cwnd$
28          $\triangleright \gamma$: EWMA parameter
29          $\triangleright \beta$: Additive Increase
30   **return** *cwnd*

Parameters: POWERTCP has only two parameters, that is the EWMA parameter γ and the additive increase parameter β. γ dictates the balance in reaction time and sensitivity to noise. We recommend $\gamma = 0.9$ based on our parameter sweep over wide range of scenarios including traffic patterns that induce

rapid fluctuations in the bottleneck queue lengths. Reflecting the intuition for additive increase in prior work [36], we set $\beta = \frac{HostBw \times \tau}{N}$ where N is the expected number of flows sharing host NIC, $HostBw$ is the NIC bandwidth at the host and τ is the base-RTT. This is to avoid queuing at the local interface or, in other words, to avoid making the host NIC a bottleneck, assuming a maximum of N flows share the host NIC bandwidth. Finally, all flows transmit at line rate in the first RTT and use $cwnd_{init} = HostBw \times \tau$. By transmitting at line rate, a new flow is able to discover the bottleneck link state and reduce its *cwnd* accordingly without getting throttled due to the presence of existing flows.

3.4 Properties of POWERTCP

POWERTCP comes with strong theoretical guarantees. We show that POWERTCP's control law achieves asymptotic stability with a unique equilibrium point that satisfies our desired equilibrium state properties (Eq. 1). POWERTCP also guarantees rapid convergence to equilibrium and achieves fairness at the same time. In the following we outline POWERTCP's properties and defer the proofs to Appendix A.

Theorem 1 (Stability). POWERTCP's *control law is Lyapunov-stable as well as asymptotically stable with a unique equilibrium point.*

Theorem 2 (Convergence). *After a perturbation,* POWERTCP's *control law exponentially converges to equilibrium with a time constant* $\frac{\delta t}{\gamma}$ *where* δt *is the window update interval.*

Theorem 3 (Fairness). POWERTCP *is* β_i *weighted proportionally fair, where* β_i *is the additive increase used by a flow i.*

Theorem 1 and Theorem 2 state the key properties of POWERTCP. First, the convergence with time constant of $\frac{\delta t}{\gamma}$ shows the fast reaction to perturbations. Second, the system being asymptotically stable at low queue lengths satisfies our stringent equilibrium property discussed in §2. Indeed, **power** and Property 1 play a key role in the proof of Theorem 1 and Theorem 2 (Appendix A) revealing its importance in congestion control. In Figure 3c, we see the trajectories of POWERTCP from different initial states to a unique equilibrium without violating throughput and latency requirements, showing the accurate control enabled by power-based congestion control.

3.5 θ-POWERTCP: Standalone Version

POWERTCP's control law requires in-network queue length information which can be obtained by using techniques such as INT. In order to widen its applicability, POWERTCP can still be deployed in datacenters with legacy, non-programmable switches through accurate RTT measurement capabilities at the end-host. In this case, we rearrange term $\frac{e}{f}$ in Eq. 7 as follows,

$$\frac{e}{f} = \frac{b^2 \cdot \tau}{\Gamma} = \frac{b^2 \cdot \tau}{(\dot{q} + b) \cdot (q + b \cdot \tau)} = \frac{\tau}{(\frac{\dot{q}}{b} + 1) \cdot (\frac{q}{b} + \tau)}$$

finally, using the fact that $\frac{q}{b} + \tau = \theta$ (RTT) and $\frac{\dot{q}}{b} = \dot{\theta}$ (RTT-gradient), we reduce $\frac{e}{f}$ to,

$$\frac{e}{f} = \frac{\tau}{(\dot{\theta}+1) \cdot (\theta)} \quad (8)$$

where $\dot{\theta}$ is the RTT-gradient and θ is RTT. Using Eq. 8 in Eq. 7 allows for deployment even when INT is not supported by switches in the datacenter. Due to space constraints we moved the algorithm to Appendix D, presenting θ-POWERTCP in Algorithm 2. This algorithm demonstrates how POWERTCP's control law can be mimicked by using a delay signal without the need for switch support. However, as we will show later in our evaluation, there are drawbacks in using RTT instead of queue lengths. First, notice how queue lengths are changed to RTT, where we assume bottleneck *txRate* (μ) as bandwidth (b). The implication is that, when using *txRate* which is essentially obtained from INT, the control law knows the exact transmission rate and rapidly fills the available bandwidth. But, when using RTT, the control law assumes the bottleneck is at maximum transmission rate and does not react by multiplicative increase and rather relies on slow additive increase to fill the available bandwidth. Secondly, in multi-bottleneck scenarios, the control law precisely reacts to the most bottlenecked link when using INT but reacts to the sum of queuing delays when using RTT. Nevertheless, under congestion, both POWERTCP and θ-POWERTCP have the same properties in a single-bottleneck scenario.

3.6 Deploying POWERTCP

Modern programmable switches are able to export user-defined header fields and device metrics [18, 32]. These metrics can be embedded into data packets, a mechanism commonly referred to as in-band network telemetry (INT). POWERTCP leverages INT to obtain fine-grained, per-packet feedback about queue occupancies, traffic counters, and link configurations within the network. For deployment with legacy networking equipment, we have proposed θ-POWERTCP which only requires accurate timestamps to measure the RTT.

We imagine POWERTCP and θ-POWERTCP to be deployed on low-latency kernel-bypass stacks such as SNAP [11] or using NIC offload. Yet, in this work, instead of implementing our algorithms for these platforms, we show how POWERTCP and θ-POWERTCP can readily be deployed by merely changing the control logic of existing congestion control algorithms. In particular, we compare our work to HPCC [36] which is based on INT feedback and SWIFT [34] which is based on delay feedback.

POWERTCP requires the same switch support and header format as HPCC, as well as packet pacing support from the NIC. Additionally, it does not maintain additional state compared to HPCC but requires one extra parameter γ, the moving average parameter for window updates. Similar to SWIFT and TIMELY, θ-POWERTCP requires accurate packet timestamps from the NIC but it does not require any switch support. The simpler logic of θ-POWERTCP (compared to POW-

ERTCP) only reacts once per RTT and reduces the number of congestion control function calls.

The core contribution of this paper is the design of a novel control law and we do not explore implementation challenges further at this point since POWERTCP does not add additional complexity compared to existing algorithms. Still, to confirm the practical feasibility of our approach, we implemented POWERTCP as a Linux kernel congestion control module. We also implemented the INT component as a proof of concept for the Intel Tofino switch ASIC [18].

The switch implementation is written in P4 and uses a direct counter associated with the egress port to maintain the so far transmitted bytes and appends this metric together with the current queue occupancy upon dequeue from the traffic manager to each segment. We leverage a custom TCP option type to encode this data and append 64 bit per-hop headers to a 32 bit base header. The implementation uses less than one out of 12 stages of the Tofino's ingress pipeline (where the headers are prepared and appended) and less than one out of 12 stages in the egress pipeline (where the measurements are taken and inserted). The processing logic runs at line rate of 3.2 Tbit per second.

4 Evaluation

We evaluate the performance of POWERTCP and θ-POWERTCP and compare against existing CC algorithms. Our evaluation aims at answering four main questions.

(Q1) How well does POWERTCP react to congestion?
We find that POWERTCP outperforms the state-of-the-art congestion control algorithms, reducing tail buffer occupancy and consequently tail latency under congestion by 30% when compared to HPCC and at least by 60% compared to TIMELY and DCQCN.

(Q2) Does POWERTCP introduce a tradeoff between throughput and latency?
Our evaluation shows that POWERTCP does not trade throughput for latency and that POWERTCP rapidly converges to near-zero queue lengths without losing throughput.

(Q3) How much can we benefit under realistic workloads?
We show that POWERTCP improves 99th-percentile flow completion times for short flows ($< 10KB$) by 33% compared to HPCC, by 99% compared to HOMA and by 74% compared to TIMELY and DCQCN even at moderate network loads. At the same time, we find that POWERTCP does not penalize long flows ($> 1MB$). In fact, we find that θ-POWERTCP performs equally well for short flows compared to POWERTCP but performs similarly to TIMELY for medium and long flows.

(Q4) How does POWERTCP perform under high load and bursty traffic patterns?
Our evaluation shows that the benefits of POWERTCP are further enhanced under high loads and that POWERTCP remains stable even under bursty traffic.
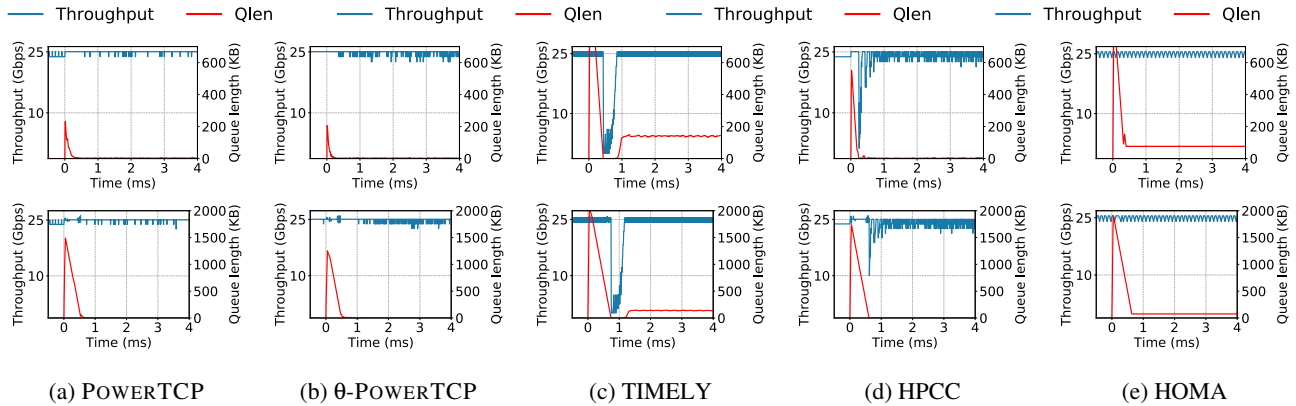
Figure 4: State-of-the-art congestion control algorithms vs POWERTCP in response to an incast. For each algorithm, we show the corresponding reaction to 10 : 1 incast in the top row and to 255 : 1 incast in the bottom row.

4.1 Setup

Our evaluation is based on network simulator NS3 [4].

Topology: We consider a datacenter network based on a Fat-Tree topology [5] with 2 core switches and 256 servers organized into four pods. Each pod consists of two ToR switches and two aggregation switches. The capacity of all the switch-to-switch links are 100Gbps and server-to-switch links are all 25Gbps leading to 4 : 1 oversubscription similar to prior work [49]. The links connecting to core switches have a propagation delay of 5μs and all the remaining links have a propagation delay of 1μs. We set up a shared memory architecture on all the switches and enable the Dynamic Thresholds algorithm [17] for buffer management across all the ports, commonly enabled in datacenter switches [1,2]. Finally we set the buffer sizes in our topology proportional to the bandwidth-buffer ratio of Intel Tofino switches [18].

Traffic mix: We generate traffic using the web search [7] flow size distribution to evaluate our algorithm using realistic workloads. We evaluate an average load (on the ToR uplinks) in the range of $20\% - 95\%$. We also use a synthetic workload similar to prior work [6] to generate incast traffic. Specifically, the synthetic workload represents a distributed file system where each server requests a file from a set of servers chosen uniformly at random from a different rack. All the servers which receive the request respond at the same time by transmitting the requested part of the file. As a result, each file request creates an incast scenario. We evaluate across different request rates and request sizes.

Comparisons and metrics: We evaluate POWERTCP with and without switch support and compare to HPCC [36], DC-QCN [56], and TIMELY [41] representing sender-based control law approaches similar to POWERTCP and HOMA [42] representing receiver-driver transport. We report flow completion times and switch buffer occupancy metrics.

Configuration: We set $\gamma = 0.9$ for POWERTCP and θ-POWERTCP. Both HPCC and POWERTCP are configured with base-RTT (τ) set to the maximum RTT in our topology and *HostBw* is set to the server NIC bandwidth. The

product of base-RTT and *HostBw* is configured as RTTBytes for HOMA and the over-commitment level is set to 1 where HOMA performed best across different overcommitment levels in our setup. We report our results for all overcommitment levels (1-6) in Appendix C. We set the parameters for DCQCN following the suggestion in [36] which is based on experience and TIMELY parameters are set according to [41].

4.2 Results

POWERTCP reacts rapidly yet accurately to congestion: We evaluate POWERTCP's reaction to congestion in two scenarios: *(i)* 10 : 1 small-scale incast and *(ii)* 255 : 1 large-scale incast. Figure 4 shows the aggregate throughput and the buffer occupancy at the bottleneck link for POWERTCP, TIMELY, HPCC and HOMA. First, at time $t = 0$, we launch ten flows simultaneously towards the receiver of a long flow leading to a **10:1** incast. We show in Figure 4a and Figure 4b that POWERTCP quickly mitigates the incast and reaches near zero queue lengths without losing throughput. In Figure 4d we see that HPCC indeed reacts quickly to get back to near-zero queue lengths. On one hand, however, HPCC does not react enough during the congestion onset and reaches higher buffer occupancy ≈ 2x compared to POWERTCP and on the other hand loses throughput after mitigating the incast as opposed to POWERTCP's stable throughput. TIMELY as shown in Figure 4c does not control the queue-lengths either and loses throughput after reacting to the incast. While HOMA sustains throughput, we observe from Figure 4e that HOMA does not accurately control bottleneck queue-lengths. Second, at time $t = 0$, in addition to the 10 : 1 incast, the 256th server sends a query request (§4.1) to all the other 255 servers which then respond at the same time, creating a **255:1** incast. From Figure 4a and Figure 4b (bottom row), we observe similar benefits from both POWERTCP and θ-POWERTCP even at large-scale incast: both react quickly and converge to near-zero queue-lengths without losing throughput. In contrast, from Figure 4c and Figure 4d we see that TIMELY and HPCC lose throughput immediately after re-

(a) POWERTCP (*ms* scale) (b) HOMA (*ms* scale)

(c) θ-POWERTCP (d) TIMELY
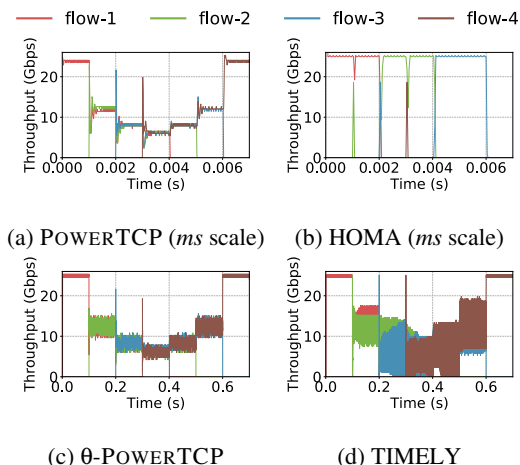
Figure 5: Fairness and stability

(a) 20% load (b) 60% load

Figure 6: 99.9 percentile flow completion times with websearch workload *(a)* even at low network load, POWERTCP outperforms existing algorithms and *(b)* as the load increases the benefits of POWERTCP are enhanced. However, only short flows benefit from θ-POWERTCP.

acting to the increased queue length. From Figure 4e we observe that HOMA reaches approximately 500KB higher queue-length compared to POWERTCP and cannot converge to near-zero queue-lengths quickly.

POWERTCP is stable and achieves fairness: POWERTCP not only reacts rapidly to reduce queue lengths but also features excellent stability. Figure 5 shows how bandwidth is shared by multiple flows as they arrive and leave. We see that POWERTCP stabilizes to a fair share of bandwidth quickly, both when flows arrive and leave, confirming POWERTCP's fast reaction to congestion as well as the available bandwidth.

Figure 4a showing convergence and Figure 5a showing fairness and stability confirm the theoretical guarantees of POWERTCP. Hereafter, all our results are based on the setup described above, §4.1, using realistic workloads.

POWERTCP significantly improves short flows FCTs: In Figure 6 we show the 99.9-percentile flow completion times using POWERTCP and state-of-the-art datacenter congestion control algorithms. At 20% network load (Figure 6a), POWERTCP and θ-POWERTCP improve 99.9-percentile flow completion times for short flows ($< 10KB$) by 9% compared to HPCC and by 80% compared to TIMELY, DCQCN and HOMA. Even at moderate load of 60% (Figure 6b), short flows significantly benefit from POWERTCP as well as θ-POWERTCP. Specifically, POWERTCP improves 99.9 percentile flow completion times for short flows by 33% compared to HPCC, by 99% compared to HOMA and by 74% compared to TIMELY and DCQCN. θ-POWERTCP provides even greater benefits to short flows showing an improvement of 36% compared to HPCC and 82% compared to TIMELY and DCQCN. Indeed, web search workload being buffer-intensive, our results confirm the observations made in §2. TIMELY being a current-based CC, does not explicitly control queuing latency, while HPCC, a voltage-based CC, does not react as fast as POWERTCP to mitigate congestion resulting in higher flow completion times. Surprisingly, HOMA
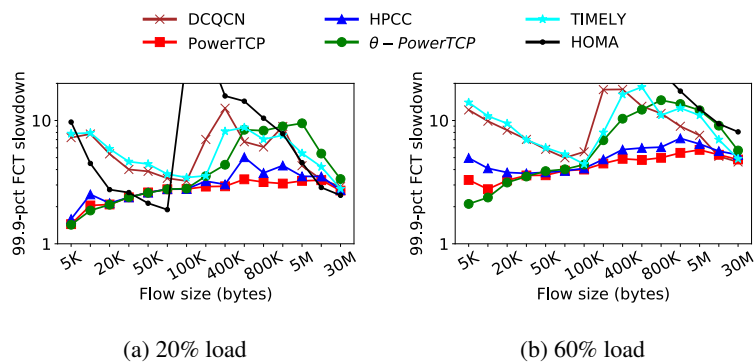
performs the worst, showing an order-of-magnitude higher FCTs for short flows at high loads as shown in Figure 6b.

We also evaluate across various loads in the range $20\% - 95\%$ and show the 99.9-percentile flow completion times for short flows in Figure 7a. In particular, we see that the benefits of POWERTCP and θ-POWERTCP are further enhanced as the network load increases. POWERTCP (and θ-POWERTCP) improve the flow completion times of short flows by 36% (and 55%) compared to HPCC. Short flows particularly benefit from POWERTCP due its accurate control of buffer occupancies close to zero. In Figure 7g we show the CDF of buffer occupancies at 80% load. POWERTCP consistently maintains lower buffer occupancy and cuts the tail buffer occupancy by 50% compared to HPCC.

Medium sized flows also benefit from POWERTCP: We find that POWERTCP not only improves short flow performance but also improves the 99.9-percentile flow completion times for medium sized flows ($100KB - 1M$). In Figure 6 we see that POWERTCP consistently achieves better flow completion times for medium sized flows. Specifically, at 20% network load (Figure 6a), POWERTCP improves 99.9-percentile flow completion times for medium flows by 33% compared to HPCC, by 76% compared to HOMA and by 62% (and 50%) compared to TIMELY (and DCQCN). In Figure 6b, we observe similar benefits even at 60% load.

We notice from Figure 6a and Figure 6b that the performance of θ-POWERTCP deteriorates sharply for medium sized flows. θ-POWERTCP uses RTT for window update calculations. While RTT can be a good congestion signal, it does not signal under-utilization as opposed to INT that explicitly notifies the exact utilization. As a result, medium flows with θ-POWERTCP experience 60% worse performance on average compared to POWERTCP and HPCC. We also observe similar performance for TIMELY that uses RTT as a congestion signal. Although delay is simple and effective for short flows performance even at the tail, our results show that delay

(a) Short flows FCT with web-search workload

(b) Long flows FCT with web-search workload

(c) Short flows FCTs with web-search + incasts

(d) Long flows FCT with web-search + incasts

(e) Short flows FCTs with web-search + incasts

(f) Long flows FCT with web-search + incasts

(g) Buffer occupancy with web-search workload at 80% load

(h) Buffer occupancy with web-search + incasts
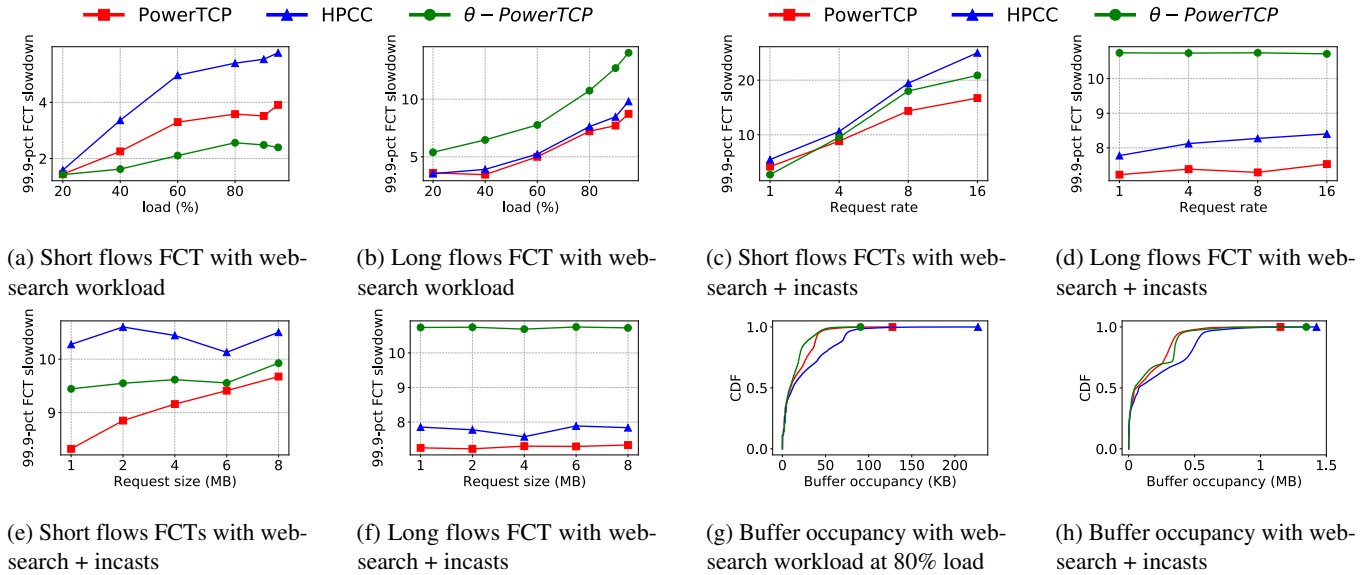
Figure 7: A detailed comparison of POWERTCP, θ-POWERTCP and the state-of-the-art showing the benefits of POWERTCP and the trade-offs of θ-POWERTCP. Particularly POWERTCP outperforms the state-of-the-art across a range of network loads even under bursty traffic. However, θ-POWERTCP performs well for short flows but long flows cannot benefit from θ-POWERTCP.

as a congestion signal is not ideal if not worse for medium sized flows.

POWERTCP does not penalize long flows: Fast reaction to available bandwidth makes POWERTCP ideal for best performance across all flow sizes. We observe from Figure 6 that POWERTCP achieves flow completion times comparable to existing algorithms, indicating that POWERTCP does not trade throughput for low latency. Further, in Figure 7b we show the 99.9-percentile flow completion times for long flows across various loads. At low load, POWERTCP performs similar to HPCC and performs 9% better compared to HPCC at 90% network loads. However, we see that θ-POWERTCP is consistently 35% worse on average across various loads compared to POWERTCP and HPCC.

POWERTCP outperforms under bursty traffic: We generate incast-like traffic described in §4.1 in addition to the web search workload at 80% load. In Figure 7c and Figure 7d we show the 99.9-percentile flow completion times for short and long flows across different request rates for a request size of 2*MB*. Note that by varying request rates, we are essentially varying the frequency of incasts. We observe that even under bursty traffic, POWERTCP improves 99.9-percentile flow completion times on average for short flows by 24% and for long flows by 10% compared to HPCC. Further POWERTCP outperforms at high request rates showing 33% improvement over HPCC for short flows. On the other hand, θ-POWERTCP improves flows completion times for short flows but performs worse across all request rates compared to HPCC.

We further vary the request size at a request rate of four per second. Note that by varying the request size, we also vary the duration of congestion. In Figure 7e and Figure 7f,

we show the 99.9-percentile flow completion times for short and long flows. Specifically, in Figure 7e we observe that flow completion times with POWERTCP gradually increase with request size. POWERTCP, compared to HPCC, improves flow completion times of short flows by 20% at 1*MB* request size and improves by 7% at 8*MB* request size. At the same time, POWERTCP does not sacrifice long flows performance under bursty traffic. POWERTCP improves flow completion times for long flows by 5% on average compared to HPCC. θ-POWERTCP's performance similar to previous experiments is on average 30% worse for long flows but 9% better for short flows compared to HPCC. We show the CDF of buffer occupancies under bursty traffic with 2*MB* request size and 16 per second request rate. Both POWERTCP and θ-POWERTCP reduce the 99 percentile buffer by 31% compared to HPCC.

We note that HOMA's performance in our evaluation is not in line with the results presented in [42]. Recent work [26] reports similar performance issues with HOMA. We suspect two possible reasons: *(i)* HOMA's accuracy in controlling congestion is specifically limited in our network setup with an oversubscribed Fat-Tree topology where congestion at the ToR uplinks is a possibility which cannot be controlled by a receiver-driven approach such as HOMA. *(ii)* As pointed out by [26], HOMA's original evaluation considered practically infinite buffers at the switches whereas switches in our setup are limited in buffer and use Dynamic Thresholds to share buffer. Further, even at 20% load, asymmetric RTTs in a Fat-Tree topology (consequently RTTBytes) across ToR pairs contributes to HOMA's inaccuracy in controlling congestion.
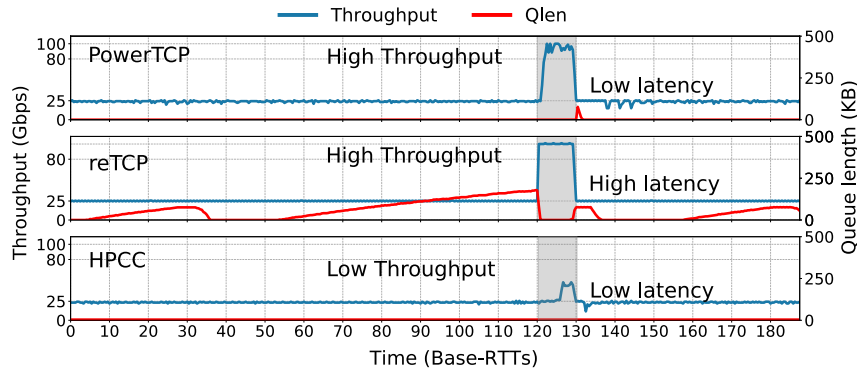
(a) POWERTCP reacts rapidly to the available bandwidth achieving good circuit utilization.
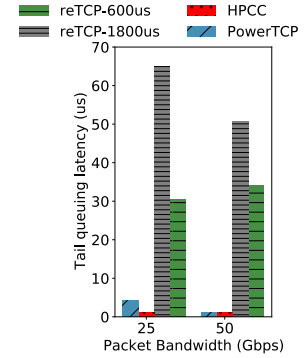
(b) POWERTCP significantly reduces the tail latency

Figure 8: The benefits of POWERTCP in reconfigurable datacenter networks showing its ability to achieve good circuit utilization while significantly reducing the tail latency compared to reTCP.

5 Case Study: Reconfigurable DCNs

Given POWERTCP's rapid reaction to congestion and available bandwidth, we believe that POWERTCP is well suited for emerging reconfigurable datacenter networks (RDCN) [44]. We now examine POWERTCP's applicability in this context through a case study. Congestion control in RDCNs is especially challenging as the available bandwidth rapidly fluctuates due to changing circuits. In this section, we evaluate the performance of POWERTCP and compare against the state-of-the-art reTCP [43] and HPCC using packet-level simulations in NS3. We implement both POWERTCP and HPCC in the transport layer and limit their window updates to once per RTT for a fair comparison with reTCP. POWERTCP and HPCC flows initialize the TCP header with the unused option number 36. Switches are configured to append INT metadata to TCP options. It should be noted that TCP options are limited to 40 bytes. As a result, our implementation can only support at most four hops round-trip path length.

We evaluate in a topology with 25 ToR switches with 10 servers each and a single optical circuit switch connected to all the ToR switches. ToR switches are also connected to a separate packet switched network with 25Gbps links. The optical switch internally connects each input port to an output port and cycles across 24 matchings in a permutation schedule where the switch stays in a specific matching for $225\mu s$ (one day) and takes $20\mu s$ to reconfigure to the next matching (one night). In this setting, each pair of ToR switches has direct connectivity through the circuit switch once over a length of 24 matchings (one week). We use single-hop routing in the circuit network and a maximum base RTT is $24\mu s$. Note that circuit-on time (i.e., one day) is approximately 10 RTTs. The links between servers and ToR switches are 25Gbps and circuit links are 100Gbps. We configure the ToR switches to forward packets exclusively on the circuit network when available. Switches are further equipped with per-destination virtual output queues (VOQs). Our setup is in line with prior work [43]. We set reTCP's prebuffering to $1800\mu s$ based

on the suggestions in [43] and set to $600\mu s$ based on our parameter sweep for the minimum required prebuffering in our topology. We compare against both versions.

In Figure 8a, we show the time series of throughput and VOQ length for a pair of ToR switches. Specifically, the gray-shaded area in Figure 8a highlights the availability of high bandwidth through the circuit-switched network. On one hand, reTCP instantly fills the available bandwidth but incurs high latency due to prebuffering before the circuit is available. On the other hand, HPCC maintains low queue lengths but does not fill the available bandwidth. In contrast, POWERTCP fills the available bandwidth within one RTT and maintains near-zero queue lengths and thereby achieves both high throughput and low latency. We show the tail queuing latency incurred by reTCP, HPCC and POWERTCP in Figure 8b. We observe that POWERTCP improves the tail queuing latency at least by 5× compared to reTCP. Our case study reveals that fine-grained congestion control algorithms such as POWERTCP can alleviate the circuit utilization problem in RDCNs without trading latency for throughput.

6 Related Work

Dealing with congestion has been an active research topic for decades with a wide spectrum of approaches, including buffer management [3, 10, 17] and scheduling [9, 25, 45, 46]. In the following, we will focus on the most closely related works on end-host congestion control.

Approaches such as [7, 51, 56] (e.g., DCTCP, D²TCP) rely on ECN as the congestion signal and react proportionally. Such algorithms require the bottleneck queue to grow up to a certain threshold, which results in queuing delays. ECN-based schemes remain oblivious to congestion onset and intensity. Protocols such as TIMELY [41], SWIFT [34], CDG [23], DX [35] rely on RTT measurements for window update calculations. TIMELY and CDG partly react to congestion based on delay gradients, remaining oblivious to absolute queue lengths. TIMELY, for instance, uses a threshold to fall back

to proportional reaction to delay instead of delay gradient. SWIFT, a successor of TIMELY, only reacts proportionally to delay. As a result, SWIFT cannot detect congestion onset and intensity unless the distance from target delay significantly increases. In contrast, θ-POWERTCP also being a delay-based congestion control algorithm updates the window sizes using the notion of power. As a result, θ-POWERTCP accurately detects congestion onset even at near-zero queue lengths.

XCP [30], D³ [52], *RCP* [19] rely on explicit network feedback based on rate calculations within the network. However, the rate calculations are based on heuristics and require parameter tuning to adjust for different goals such as fairness and utilization. HPCC [36] introduces a novel use of in-band network telemetry and significantly improves the fidelity of feedback. Our work builds on the same INT capabilities to accurately measure the bottleneck link state. However, as we show analytically and empirically, HPCC's control law then adjusts rate and window size solely based on observed queue lengths and lacks control accuracy compared to POWERTCP. Our proposal POWERTCP uses the same feedback signal but uses the notion of power to update window sizes leading to significantly more fine-grained and accurate reactions.

Receiver-driven transport protocols such as NDP [22], HOMA [42], and Aeolus [26] have received much attention lately. Such approaches are conceptually different from classic transmission control at the sender. Importantly, receiver-driven transport approaches make assumptions on the uniformity in datacenter topologies and oversubscription [22]. POWERTCP is a sender-based classic CC approach that uses our novel notion of power and achieves fine-grained control over queuing delays without sacrificing throughput.

7 Conclusion

We presented POWERTCP, a novel fine-grained congestion control algorithm. By reacting to both the current state of the network as well as its trend (i.e., power), POWERTCP improves throughput, reduces latency, and keeps queues within the network short. We proved that POWERTCP has a set of desirable properties, such as fast convergence and stability allowing it to significantly improve flow completion times compared to the state-of-the-art. Its fast reaction makes POWERTCP attractive for many dynamic network environments including emerging reconfigurable datacenters which served us as a case study in this paper. In our future work, we plan to explore more such use cases.

References

[1] Broadcom. 12.8 tb/s strataxgs tomahawk 3 ethernet switch series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56980-series.

[2] Broadcom. 2020. 25.6 tb/s strataxgs tomahawk 4 ethernet switch series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56990-series.

[3] Cisco nexus 9000 series switches. https://www.cisco.com/c/en/us/products/collateral/switches/nexus-9000-series-switches/white-paper-c11-738488.html.

[4] Ns3 network simulator. https://www.nsnam.org/.

[5] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference*, page 63–74, 2008.

[6] Mohammad Alizadeh and Tom Edsall. On the data path performance of leaf-spine datacenter fabrics. In *2013 IEEE 21st annual symposium on high-performance interconnects*, pages 71--74. IEEE, 2013.

[7] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 63--74, 2010.

[8] Mohammad Alizadeh, Adel Javanmard, and Balaji Prabhakar. Analysis of dctcp: stability, convergence, and fairness. *ACM SIGMETRICS Performance Evaluation Review*, 39(1):73--84, 2011.

[9] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. Pfabric: Minimal near-optimal datacenter transport. In *Proceedings of the ACM SIGCOMM 2013 Conference*, page 435–446, 2013.

[10] Maria Apostolaki, Laurent Vanbever, and Manya Ghobadi. Fab: Toward flow-aware buffer sharing on programmable switches. In *Proceedings of the 2019 Workshop on Buffer Sizing*, pages 1--6, 2019.

[11] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. Snap: Stateful network-wide abstractions for packet processing. In *Proceedings of the ACM SIGCOMM 2016 Conference*, page 29–43, 2016.

[12] Chen Avin, Manya Ghobadi, Chen Griner, and Stefan Schmid. On the complexity of traffic traces and implications. In *Proc. ACM SIGMETRICS*, 2020.

[13] Chen Avin and Stefan Schmid. Renets: Statically-optimal demand-aware networks. In *Proc. SIAM Symposium on Algorithmic Principles of Computer Systems (APOCS)*, 2021.

[14] Hitesh Ballani, Paolo Costa, Raphael Behrendt, Daniel Cletheroe, Istvan Haller, Krzysztof Jozwik, Fotini Karinou, Sophie Lange, Kai Shi, Benn Thomsen, and Hugh Williams. Sirius: A flat datacenter network with nanosecond optical switching. In *Proceedings of the ACM SIGCOMM 2020 Conference*, page 782–797, 2020.

[15] Lawrence S Brakmo, Sean W O'Malley, and Larry L Peterson. Tcp vegas: New techniques for congestion detection and avoidance. In *Proceedings of the conference on Communications architectures, protocols and applications*, pages 24--35, 1994.

[16] Yanpei Chen, Rean Griffith, Junda Liu, Randy H Katz, and Anthony D Joseph. Understanding tcp incast throughput collapse in datacenter networks. In *Proceedings of the 1st ACM workshop on Research on enterprise networking*, pages 73--82, 2009.

[17] Abhijit K Choudhury and Ellen L Hahne. Dynamic queue length thresholds for shared-memory packet switches. *IEEE/ACM Transactions On Networking*, 6(2):130--140, 1998.

[18] Intel Corporation. Intel Tofino, 2020. Retrieved Dec. 29, 2020 from https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html.

[19] Nandita Dukkipati and Nick McKeown. Why flow-completion time is the right metric for congestion control. *ACM SIGCOMM Computer Communication Review*, 36(1):59--62, 2006.

[20] Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Nikhil Devanur, Janardhan Kulkarni, Gireeja Ranade, Pierre-Alexandre Blanche, Houman Rastegarfar, Madeleine Glick, and Daniel Kilper. Projector: Agile reconfigurable data center interconnect. In *Proceedings of the ACM SIGCOMM 2016 Conference*, pages 216--229, 2016.

[21] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS operating systems review*, 42(5):64--74, 2008.

[22] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the ACM SIGCOMM 2017 Conference*, page 29–42, 2017.

[23] David A Hayes and Grenville Armitage. Revisiting tcp congestion control using delay gradients. In *International Conference on Research in Networking*, pages 328--341. Springer, 2011.

[24] Christopher V Hollot, Vishal Misra, Don Towsley, and Wei-Bo Gong. A control theoretic analysis of red. In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No. 01CH37213)*, volume 3, pages 1510--1519. IEEE, 2001.

[25] Chi-Yao Hong, Matthew Caesar, and P Brighten Godfrey. Finishing flows quickly with preemptive scheduling. *ACM SIGCOMM Computer Communication Review*, 42(4):127--138, 2012.

[26] Shuihai Hu, Wei Bai, Gaoxiong Zeng, Zilong Wang, Baochen Qiao, Kai Chen, Kun Tan, and Yi Wang. Aeolus: A building block for proactive transport in datacenters. In *Proceedings of the ACM SIGCOMM 2020 Conference*, page 422–434, 2020.

[27] V. Jacobson. Congestion avoidance and control. In *Symposium Proceedings on Communications Architectures and Protocols*, SIGCOMM '88, page 314–329, 1988.

[28] Cheng Jin, David X Wei, and Steven H Low. Fast tcp: motivation, architecture, algorithms, performance. In *IEEE INFOCOM 2004*, volume 4, pages 2490--2501. IEEE, 2004.

[29] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. The nature of data center traffic: measurements & analysis. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement*, pages 202--208, 2009.

[30] Dina Katabi, Mark Handley, and Charlie Rohrs. Congestion control for high bandwidth-delay product networks. In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 89--102, 2002.

[31] Srinivasan Keshav. *Mathematical foundations of computer networking*. Addison-Wesley, 2012.

[32] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. In-band network telemetry via programmable dataplanes. In *ACM SIGCOMM '15 Demos*, 2015.

[33] Janardhan Kulkarni, Stefan Schmid, and Pawel Schmidt. Scheduling opportunistic links in two-tiered reconfigurable datacenters. In *33rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2021.

[34] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proceedings of the ACM SIGCOMM 2020 Conference*, page 514–528, 2020.

[35] Changhyun Lee, Chunjong Park, Keon Jang, Sue Moon, and Dongsu Han. Accurate latency-based congestion feedback for datacenters. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 403--415, Santa Clara, CA, July 2015. USENIX Association.

[36] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. Hpcc: High precision congestion control. In *Proceedings of the ACM SIGCOMM 2019 Conference*, pages 44--58, 2019.

[37] S.H. Low, F. Paganini, and J.C. Doyle. Internet congestion control. *IEEE Control Systems Magazine*, 22(1):28--43, 2002.

[38] William M Mellette, Rajdeep Das, Yibo Guo, Rob McGuinness, Alex C Snoeren, and George Porter. Expanding across time to deliver bandwidth efficiency and low latency. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 1--18, 2020.

[39] William M Mellette, Rob McGuinness, Arjun Roy, Alex Forencich, George Papen, Alex C Snoeren, and George Porter. Rotornet: A scalable, low-complexity, optical datacenter network. In *Proceedings of the ACM SIGCOMM 2017 Conference*, pages 267--280, 2017.

[40] Vishal Misra, Wei-Bo Gong, and Don Towsley. Fluid-based analysis of a network of aqm routers supporting tcp flows with an application to red. In *Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 151--160, 2000.

[41] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. In *Proceedings of the ACM SIGCOMM 2015 Conference*, page 537–550, 2015.

[42] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the ACM SIGCOMM 2018 Conference*, page 221–235, 2018.

[43] Matthew K Mukerjee, Christopher Canel, Weiyang Wang, Daehyeok Kim, Srinivasan Seshan, and Alex C Snoeren. Adapting TCP for reconfigurable datacenter networks. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 651--666, 2020.

[44] Matthew Nance Hall, Klaus-Tycho Foerster, Stefan Schmid, and Ramakrishnan Durairajan. A survey of reconfigurable optical networks. *Optical Switching and Networking*, 41:100621, 2021.

[45] Jonathan Perry, Hari Balakrishnan, and Devavrat Shah. Flowtune: Flowlet control for datacenter networks. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 421--435, Boston, MA, March 2017. USENIX Association.

[46] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A centralized" zero-queue" datacenter network. In *Proceedings of the ACM SIGCOMM 2014 conference*, pages 307--318, 2014.

[47] Amar Phanishayee, Elie Krevat, Vijay Vasudevan, David G Andersen, Gregory R Ganger, Garth A Gibson, and Srinivasan Seshan. Measurement and analysis of tcp throughput collapse in cluster-based storage systems. In *FAST*, volume 8, pages 1--14, 2008.

[48] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. Inside the social network's (datacenter) network. In *Proceedings of the ACM SIGCOMM 2015 Conference*, pages 123--137, 2015.

[49] Ahmed Saeed, Varun Gupta, Prateesh Goyal, Milad Sharif, Rong Pan, Mostafa Ammar, Ellen Zegura, Keon Jang, Mohammad Alizadeh, Abdul Kabbani, and Amin Vahdat. Annulus: A dual congestion control loop for datacenter and wan traffic aggregates. In *Proceedings of the ACM SIGCOMM 2020 Conference*, page 735–749, 2020.

[50] Stefan Schmid, Chen Avin, Christian Scheideler, Michael Borokhovich, Bernhard Haeupler, and Zvi Lotker. Splaynet: Towards locally self-adjusting networks. *IEEE/ACM Transactions on Networking (ToN)*, 2016.

[51] Balajee Vamanan, Jahangir Hasan, and T.N. Vijaykumar. Deadline-aware datacenter tcp (d2tcp). In *Proceedings of the ACM SIGCOMM 2012 Conference*, page 115–126, 2012.

[52] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better never than late: Meeting deadlines in datacenter networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, page 50–61, 2011.

[53] Jackson Woodruff, Andrew W Moore, and Noa Zilberman. Measuring burstiness in data center applications. In *Proceedings of the 2019 Workshop on Buffer Sizing*, 2019.

[54] Doron Zarchy, Radhika Mittal, Michael Schapira, and Scott Shenker. Axiomatizing congestion control. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 3(2):1--33, 2019.

[55] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. High-resolution measurement of data center microbursts. In *Proceedings of the 2017 Internet Measurement Conference*, pages 78--85, 2017.

[56] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. *ACM SIGCOMM Computer Communication Review*, 45(4):523--536, 2015.

[57] Yibo Zhu, Monia Ghobadi, Vishal Misra, and Jitendra Padhye. Ecn or delay: Lessons learnt from analysis of dcqcn and timely. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pages 313--327, 2016.

A Analysis

Our analysis is based on a a single bottleneck link model widely used in the literature [24, 40, 54, 57]. Specifically, we assume that all senders use the same protocol, transmit long flows sharing a common bottleneck link with bandwidth b, and have a base round trip time τ (excluding queuing delays). We denote at time t queue length as $q(t)$, aggregate window size as $w(t)$, window size of a sender i as $w_i(t)$, forward propagation delay between sender and bottleneck queue as t^f, the round-trip time as $\theta(t)$ and a base round-trip time as τ. Here $w(t) = \sum_i w_i(t)$.

We additionally use the traditional model of queue length dynamics which is independent of the control law [24, 40]

$$\dot{q}(t) = \frac{w(t - t^f)}{\theta(t)} - b \tag{9}$$

where $\theta(t)$ is given by,

$$\theta(t) = \frac{q(t)}{b} + \tau \tag{10}$$

Power at time t denoted by $\Gamma(t)$ as defined in §3.1 is expressed as,

$$\Gamma(t) = \underbrace{(q(t) + b \cdot \tau)}_{voltage} \cdot \underbrace{(\dot{q}(t) + \mu(t))}_{current} \tag{11}$$

POWERTCP's control law at a source i is given by,

$$w_i(t + \delta t) = \gamma \cdot \left(\frac{w_i(t - \theta(t)) \cdot e}{f(t)} + \beta \right) + (1 - \gamma) \cdot w_i(t) \tag{12}$$

where e and $f(t)$ are given by,

$$e = b^2 \cdot \tau$$

$$f(t) = \Gamma(t - \theta(t) + t^f)$$

and β is the additive increase term and $\gamma \in (0, 1]$ serves as the weight given for new updates using EWMA. Both β and γ are parameters to the control law.

Using the properties of power (Property 1), the aggregate window size at time $t - \theta(t)$ can be expressed in terms of power as,

$$w(t - \theta(t)) = \frac{\Gamma(t - \theta(t) + t^f)}{b} = \frac{f(t)}{b} \tag{13}$$

Suppose an *ack* arrives at time t acknowledging a segment, time $t - \theta(t)$ corresponds to the time when the acknowledged segment was transmitted.

Theorem 1 (Stability)**.** POWERTCP*'s control law is Lyapunov-stable as well as asymptotically stable with a unique equilibrium point.*

Notation	Description
b	bottleneck bandwidth
q	bottleneck queue length
τ	base RTT
t^f	sender to bottleneck delay
θ	round trip time RTT
w_i	window size of a flow i
w	aggregate window size (of all flows)
γ	EWMA parameter
β	additive increase
e	desired equilibrium point
f	feedback
λ_i	sending rate of a flow i
λ	Current: aggregate sending rate
ν	Voltage
Γ	Power

Table 2: Key notations used in this paper. Additionally for any variable say x, $\dot{x}$ denotes its derivative with respect to time i.e., $\frac{dx}{dt}$.

Proof. First, we rewrite Eq. 12 as follows to obtain the aggregate window w,

$$\sum_i w_i(t+\delta t) = \sum_i \gamma \cdot \left(\frac{w_i(t-\theta(t)) \cdot e}{f(t)} + \beta \right) + \sum_i (1-\gamma) \cdot w_i(t)$$

let $\hat{\beta} = \sum_i \beta$

$$w(t+\delta t) = \gamma \cdot \left(\frac{w(t-\theta(t)) \cdot e}{f(t)} + \hat{\beta} \right) + (1-\gamma) \cdot w(t)$$

by rearranging the terms in the above equation we obtain,

$$w(t+\delta t) - w(t) = \gamma \cdot \left(-w(t) + \frac{w(t-\theta(t)) \cdot e}{f(t)} + \hat{\beta} \right)$$

dividing by δt on both sides in the above equation and using Euler's first-order approximation, we derive the window dynamics for POWERTCP as follows,

$$\dot{w}(t) = \gamma_r \cdot \left(-w(t) + \frac{w(t-\theta(t)) \cdot e}{f(t)} + \hat{\beta} \right) \quad (14)$$

where $\gamma_r = \frac{\gamma}{\delta t}$. Using Eq. 13 and substituting $e = b^2 \cdot \tau$, Eq. 14 reduces to,

$$\dot{w}(t) = \gamma_r \cdot \left(-w(t) + b \cdot \tau + \hat{\beta} \right) \quad (15)$$

In the system defined by Eq. 9 and Eq. 14, when the window and the queue length stabilize i.e., $\dot{w}(t) = 0$ and $\dot{q}(t) = 0$, it is easy to observe that there exists a unique equilibrium point $(w_e, q_e) = (b \cdot \tau + \hat{\beta}, \hat{\beta})$. We now apply a change of variable from t to $t - t^f$ in Eq. 15 and linearize Eq. 15 and Eq. 9 around (w_e, q_e),

$$\delta \dot{w}(t - t^f) = -\gamma_r \cdot \delta w(t - t^f) \quad (16)$$

$$\delta \dot{q}(t) = -\frac{\delta q(t)}{\tau} + \frac{\delta w(t - t^f)}{\tau} \quad (17)$$

We now convert the above differential equations to matrix form,

$$\begin{bmatrix} \delta \dot{q}(t) \\ \delta \dot{w}(t) \end{bmatrix} = \begin{bmatrix} -\frac{1}{\tau} & \frac{1}{\tau} \\ 0 & -\gamma_r \end{bmatrix} \times \begin{bmatrix} \delta q(t) \\ \delta w(t) \end{bmatrix}$$

It is then easy to observe that the eigenvalues of the system are $-\frac{1}{\tau}$ and $-\gamma_r$. Since τ (base RTT) and $\gamma_r = \frac{\gamma}{\delta t}$ are both positive, we see that both the eigenvalues are negative. This proves that the system is both lyapunav stable and asymptotically stable. □

Theorem 2 (Convergence). *After a perturbation, POWERTCP's control law exponentially converges to equilibrium with a time constant $\frac{\delta t}{\gamma}$ where δt is the window update interval.*
Proof. A perturbation at time $t = 0$ causes the window to shift from $w_e = c \cdot \tau + \hat{\beta}$ to say w_{init}. We solve the differential equation in Eq. 15 and obtain the following equation,

$$w(t) = w_e + \underbrace{(w_{init} - w_e) \cdot e^{-\gamma_r \cdot t}}_{\text{exponential decay}} \quad (18)$$

From Eq. 18 we can see that, for any error $e = w_e - w_{init}$ caused by a perturbation, e exponentially decays with a time constant $\frac{1}{\gamma_r} = \frac{\delta t}{\gamma}$. Hence for e to decay 99.3%, it takes $\frac{5 \cdot \delta}{\gamma}$ time. □

Theorem 3 (Fairness). *POWERTCP is β_i weighted proportionally fair, where β_i is the additive increase used by a flow i.*
Proof. Recall that POWERTCP's control law for each flow i is defined as,

$$w_i(t+\delta t) = \gamma \cdot \left(\frac{w_i(t-\theta(t)) \cdot e}{f(t)} + \beta_i \right) + (1-\gamma) \cdot w_i(t)$$

From the proof of Theorem 1, we know that the equilibrium point for aggregate window size and queue length is $(w_e, q_e) = (b \cdot \tau + \hat{\beta}, \hat{\beta})$. Using this equilibrium we can also obtain the equilibrium value for $f(t)$ as,

$$f_e = (\hat{\beta} + b \cdot \tau) \cdot b$$

We can then show that w_i has an equilibrium point.

$$(w_i)_e = \frac{\hat{\beta} + b \cdot \tau}{\hat{\beta}} \cdot \beta_i$$

We use the argument that window sizes and rates are synonymous especially that POWERTCP uses pacing with rate $r_i = \frac{w_i}{\tau}$. We can then easily observe that the rate allocation is approximately max-min fair if β_i are small enough but β_i proportionally fair in general. □

B Justifying the Simplified Model

We considered a simplified control law model to study existing control laws in §2. Here we justify how the simplified model approximately captures the existing control laws. Our simplified model for congestion window update at time $t + \delta t$ is defined in Eq. 19 as a function of current congestion window size, a target e, the feedback $f(t)$, an additive increase β and an exponential moving average parameter γ.

$$w_i(t+\delta t) = \gamma \cdot \underbrace{\underbrace{\left(w_i(t) \cdot \frac{e}{f(t)} + \beta \right)}_{update} + (1-\gamma) \cdot w_i(t)}_{EWMA} \quad (19)$$

where e and $f(t)$ are given by,

$$e = \begin{cases} b \cdot \tau & \text{queue-length based CC} \\ \tau & \text{delay-based CC} \\ 1 & \text{RTT-gradient based CC} \end{cases} \quad (20)$$

$$f(t) = \begin{cases} q(t - \theta(t) + t^f) + b \cdot \tau & \text{queue-length based CC} \\ \frac{q(t-\theta(t)+t^f)}{b} + \tau & \text{delay-based CC} \\ \frac{\dot{q}(t-\theta(t)+t^f)}{b} + 1 & \text{RTT-gradient based CC} \end{cases}$$
$$(21)$$

We first use Euler's first order approximation and obtain the aggregate window $(\sum w)$ dynamics for the simplified model,

$$\dot{w}(t) = \frac{\gamma}{\delta t} \cdot \left(w(t) \cdot \frac{e}{f(t)} - w(t) + \beta \right) \quad (22)$$

In order for the system to stabilize, we require $\dot{q}(t) = 0$ and $\dot{w}(t) = 0$. Using Eq. 9 and Eq. 22 and applying equilibrium conditions and assuming that $f(t)$ stabilizes,

$$q_e = w_e - b \cdot \tau \quad (23)$$

$$w_e = \frac{\hat{\beta}}{1 - \frac{e}{f}} \quad (24)$$

Recall that $\hat{\beta} = \sum \beta_i$, the sum of additive increase terms of all flows sharing a bottleneck. To show whether there exists a unique equilibrium point, it remains to show whether Eq. 23 and Eq. 24 have a unique solution for w_e and q_e. We now show how the simplified model captures existing control laws and show the equilibrium properties.

Queue length or inflight-based control law: Substituting $e = b \cdot \tau$ and $f(t) = q(t - \theta(t) + t^f) + b \cdot \tau$, we express the simplified queue length based control law as,

$$w_i(t+\delta t) = \gamma \cdot \left(\frac{w_i(t) \cdot b \cdot \tau}{q(t-\theta(t)+t^f) + b \cdot \tau} + \beta \right) + (1-\gamma) \cdot w_i(t)$$
$$(25)$$

notice that the update is an MIMD based on inflight bytes. Eq. 25 captures control laws based on inflight bytes; for example HPCC [36].

A system defined by queue length based control law (Eq. 25 and the queue length dynamics (Eq. 9, there exists a unique equilibrium point. It can be observed that Eq. 24 for queue length based control law gives $w_e = b \cdot \tau + \hat{\beta}$ and $q_e = \hat{\beta}$.

Delay-based control law: Substituting $e = \tau$ and $f(t) = \frac{q(t-\theta(t)+t^f)}{b} + \tau$, we express the simplified delay-based control law as,

$$w_i(t+\delta t) = \gamma \cdot \left(\frac{w_i(t) \cdot \tau}{\frac{q(t-\theta(t)+t^f)}{b} + \tau} + \beta \right) + (1-\gamma) \cdot w_i(t) \quad (26)$$

where the window update is an MIMD based on RTT. Eq. 26 captures control laws based on RTT; for example FAST [28].

Similar to queue-length based CC, a system defined by delay-based control law (Eq. 26 and the queue length dynamics (Eq. 9, there exists a unique equilibrium point. It can be observed that Eq. 24 for delay-based control law gives $w_e = b \cdot \tau + \hat{\beta}$ and $q_e = \hat{\beta}$.

RTT-gradient based control law: Substituting $e = 1$ and $f(t) = \frac{\dot{q}(t-\theta(t)+t^f)}{b} + 1$, we express the simplified RTT-gradient based control law as,

$$w_i(t+\delta t) = \gamma \cdot \left(\frac{w_i(t) \cdot 1}{\frac{\dot{q}(t-\theta(t)+t^f)}{b} + 1} + \beta \right) + (1-\gamma) \cdot w_i(t) \quad (27)$$

where the window update is an MIMD based on RTT-gradient. Eq. 27 by rearranging the terms, captures control laws based on RTT-gradient such as TIMELY [41].

In contrast to queue-length and delay-based CC, RTT-gradient based CC has no unique equilibrium point since $f(t) = \frac{\dot{q}(t-\theta(t)+t^f)}{b} + 1$ stabilizes when $\dot{q} = 0$. However only $\dot{q} = 0$ leads to window dynamics Eq. 27 also to stabilize $(\dot{w} = 0)$ at any queue lengths. As a result under RTT-gradient control law, Eq. 23 and Eq. 24 do not have a unique solution and consequently we can state that RTT-gradient based CC has no unique equilibrium point.

C HOMA's Overcommitment

(a) Over-Commitment: 1

(b) Over-Commitment: 2

(c) Over-Commitment: 3

(d) Over-Commitment: 4

(e) Over-Commitment: 5
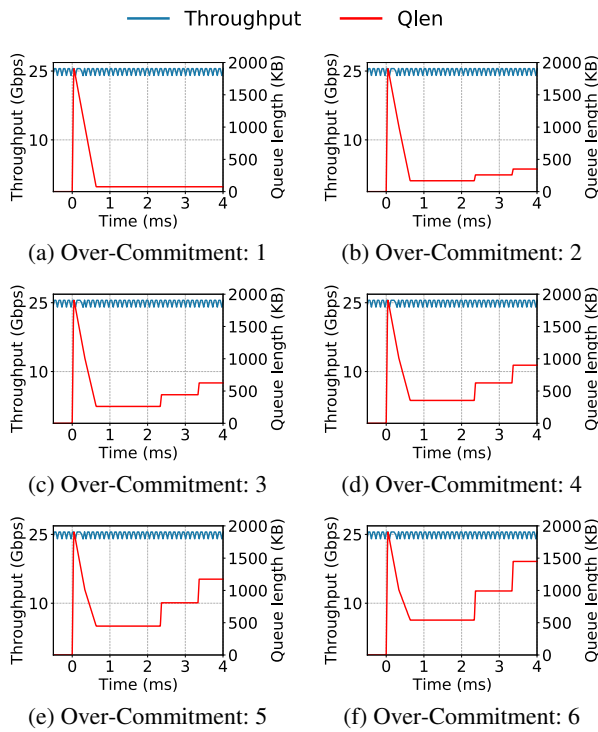
(f) Over-Commitment: 6

Figure 9: HOMA's reaction to 255 : 1 incast at different over-commitment levels.

(a) Over-Commitment: 1

(b) Over-Commitment: 2

(c) Over-Commitment: 3

(d) Over-Commitment: 4

(e) Over-Commitment: 5
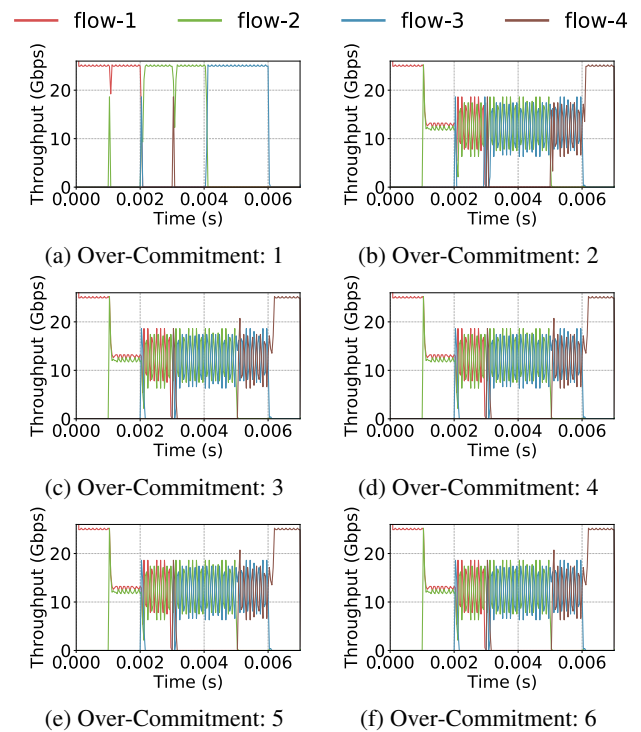
(f) Over-Commitment: 6

Figure 11: HOMA's fairness at different over-commitment levels.

(a) Over-Commitment: 1

(b) Over-Commitment: 2

(c) Over-Commitment: 3

(d) Over-Commitment: 4

(e) Over-Commitment: 5
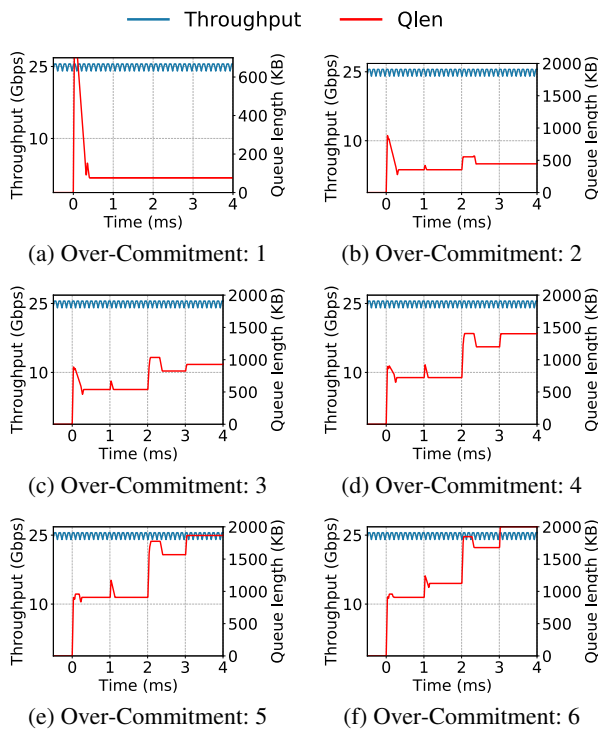
(f) Over-Commitment: 6

Figure 10: HOMA's reaction to 10 : 1 incast at different over-commitment levels.

D θ-POWERTCP

We present θ-POWERTCP: standalone version of POW-ERTCP which does not require switch support and only requires accurate packet timestamp support at the end-host.

Algorithm 2: θ-POWERTCP (w/o switch support)

---

1 /* t_c is the timestamp upon ack arrival   */
   Input : *ack*
   Output: *cwnd*, *rate*
2 **procedure** NEWACK(*ack*):
3    $cwnd_{old} = \text{GETCWND}(ack.seq)$
4    $normPower = \text{NORMPOWER}(ack)$
5    $\text{UPDATEWINDOW}(normPower, cwnd_{old})$
6    $rate = \frac{cwnd}{\tau}$
7    $prevRTT = RTT$
8    $t_c^{prev} = t_c$
9    $\text{UPDATEOLD}(cwnd, ack.seq)$
10 **function** NORMPOWER(*ack*):
11    $dt = t_c - t_c^{prev}$
12    $\dot{\theta} = \frac{RTT - prevRTT}{dt}$ ▷ $\frac{dRTT}{dt}$
13    $\Gamma_{norm} = \frac{(\dot{\theta}+1) \times RTT}{\tau}$ ▷ Γ_{norm} :Normalized power
14    $\Gamma_{smooth} = \frac{\Gamma_{smooth} \cdot (\tau - \Delta t) + \Gamma_{norm} \cdot \Delta t}{\tau}$
15   **return** Γ_{smooth}
16 **function** UPDATEWINDOW(*power, ack*):
17   **if** *ack.seq < lastUpdated* **then**   ▷ per RTT
18   **return** *cwnd*
19   **end if**
20    $cwnd = \gamma \times (\frac{cwnd_{old}}{normPower} + \beta) + (1 - \gamma) \times cwnd$
21   ▷ γ : EWMA parameter
22   ▷ β: Additive Increase
23    $lastUpdated = snd\_nxt$
24   **return** *cwnd*

RDMA is Turing complete, we just did not know it yet!

Waleed Reda
Université catholique de Louvain
KTH Royal Institute of Technology

Marco Canini
KAUST

Dejan Kostić
KTH Royal Institute of Technology

Simon Peter
University of Washington

Abstract

It is becoming increasingly popular for distributed systems to exploit offload to reduce load on the CPU. Remote Direct Memory Access (RDMA) offload, in particular, has become popular. However, RDMA still requires CPU intervention for complex offloads that go beyond simple remote memory access. As such, the offload potential is limited and RDMA-based systems usually have to work around such limitations.

We present RedN, a principled, practical approach to implementing complex RDMA offloads, without requiring any hardware modifications. Using *self-modifying* RDMA chains, we lift the existing RDMA verbs interface to a Turing complete set of programming abstractions. We explore what is possible in terms of offload complexity and performance with a commodity RDMA NIC. We show how to integrate these RDMA chains into applications, such as the Memcached key-value store, allowing us to offload complex tasks such as key lookups. RedN can reduce the latency of key-value get operations by up to $2.6\times$ compared to state-of-the-art KV designs that use one-sided RDMA primitives (e.g., FaRM-KV), as well as traditional RPC-over-RDMA approaches. Moreover, compared to these baselines, RedN provides performance isolation and, in the presence of contention, can reduce latency by up to $35\times$ while providing applications with failure resiliency to OS and process crashes.

1 Introduction

As server CPU cycles become an increasingly scarce resource, offload is gaining in popularity [23, 28, 30–32, 36]. System operators wish to reserve CPU cycles for application execution, while common, oft-repeated operations may be offloaded. NIC offloads, in particular, have the benefit that they reside in the network data path and NICs can carry out operations on in-flight data with low latency [31].

For this reason, remote direct memory access (RDMA) [15] has become ubiquitous [20]. Mellanox ConnectX NICs [4] have pioneered ubiquitous RDMA support and Intel has added RDMA support to their 800 series of Ethernet network adapters [7]. RDMA focuses on the offload of simple message passing (via SEND/RECV verbs) and remote memory access (via READ/WRITE verbs) [15]. Both primitives are widely used in networked applications and their offload is extremely useful. However, RDMA is not designed for more complex offloads that are also common in networked applications. For example, remote data structure traversal and hash table access are not normally deemed realizable with RDMA [39]. This led to many RDMA-based systems requiring multiple network round-trips or to reintroduce involvement of the server's CPU to execute such requests [18, 22, 26, 27, 35, 37, 41].

To support complex offloads, the networking community has developed a number of SmartNIC architectures [2, 3, 11, 14, 17]. SmartNICs incorporate more powerful compute capabilities via CPUs or FPGAs. They can execute arbitrary programs on the NIC, including complex offloads. However, these SmartNICs are not ubiquitous and their smaller volume implies a higher cost. SmartNICs can cost up to $5.7\times$ more than commodity RDMA NICs (RNICs) at the same link speed (§2.1). Due to their custom architecture, they are also a management burden to the system operator, who has to support SmartNICs apart from the rest of the fleet.

We ask whether we can avoid this tradeoff and attempt to use the ubiquitous RNICs to realize complex offloads. To do so, we have to solve a number of challenges. First, we have to answer if and how we can use the RNIC interface, which consists only of simple data movement verbs (READ, WRITE, SEND, RECV, etc.) and no conditionals or loops, to realize complex offloads. Our solution has to be general so that offload developers can use it to build complex *RDMA programs* that can perform a wide range of functionality. Second, we have to ensure that our solution is efficient and that we understand the performance and performance variability properties of using RNICs for complex offloads. Finally, we have to answer how complex RNIC offloads integrate with existing applications.

In this paper, we show that RDMA is *Turing complete*, making it possible to use RNICs to implement complex offloads. To do so, we implement conditional branching via *self-modifying* RDMA verbs. Clever use of the existing compare-

and-swap (CAS) verb enables us to dynamically modify the RNIC execution path by editing subsequent verbs in an RDMA program, using the CAS operands as a predicate. Just like self-modifying code executing on CPUs, self-modifying verbs require careful control of the execution path to avoid consistency issues due to RNIC verb prefetching. To do so, we rely on the WAIT and ENABLE RDMA verbs [28, 34] that provide execution dependencies. WAIT allows us to halt execution of new verbs until past verbs have completed, providing strict ordering among RDMA verbs. By controlling verb prefetching, ENABLE enforces consistency for verbs modified by preceding verbs. ENABLE also allows us to create loops by re-triggering earlier, already-executed verbs in an RDMA work queue—allowing the NIC to operate autonomously without out CPU intervention.

Based on these primitives, we present RedN, a principled, practical approach to implementing complex RNIC offloads. Using self-modifying RDMA programs, we develop a number of building blocks that lift the existing RDMA verbs interface to a Turing complete set of programming abstractions. Using these abstractions, we explore what is possible in terms of offload complexity and performance with just a commodity RNIC. We show how to integrate complex RNIC offloads, developed with RedN principles, into existing networked applications. RedN affords offload developers a practical way to implement complex NIC offloads on commodity RNICs, without the burden of acquiring and maintaining SmartNICs. Our code is available at: https://redn.io.

We make the following contributions:

• We present RedN, a principled, practical approach to offloading arbitrary computation to RDMA NICs. RedN leverages RDMA ordering and compare-and-swap primitives to build conditionals and loops. We show that these primitives are sufficient to make RDMA Turing complete.

• Using RedN, we present and evaluate the implementation of various offloads that are useful in common server computing scenarios. In particular, we implement hash table lookup with Hopscotch hashing and linked list traversal.

• We evaluate the complexity and performance of offload in a number of use cases with the Memcached key-value store. In particular, we evaluate offload of common key-value get operations, as well as performance isolation and failure resiliency benefits. We demonstrate that RNIC offload with RedN can realize all of these benefits. It can reduce average latency of get operations by up to $2.6\times$ compared to state-of-the-art one-sided RDMA key-value stores (e.g., FaRM-KV [22]), as well as traditional two-sided RPC-over-RDMA implementations. Moreover, RedN provides superior performance isolation, improving latency by up to $35\times$ under contention, while also providing higher availability under host-side failures.

2 Background

RDMA was conceived for high-performance computing (HPC) clusters, but it has grown out of this niche [20]. It

is becoming ever-more popular due to the growth in network bandwidth, with stagnating growth in CPU performance, making CPU cycles an increasingly scarce resource that is best reserved to running application code. With RNICs now considered commodity, it is opportunistic to explore the use-cases where their hardware can yield benefits. These efforts, however, have been limited by the RDMA API, which constrains the expression of many complex offloads. Consequently, the networking community has built SmartNICs using FPGAs and CPUs to investigate new complex offloads.

2.1 SmartNICs

To enable complex network offloads, SmartNICs have been developed [1, 2, 10, 11]. SmartNICs include dedicated computing units or FPGAs, memory, and several dedicated accelerators, such as cryptography engines. For example, Mellanox BlueField [11] has $8\times$ARMv8 cores with 16GB of memory and $2\times$25GbE ports. These SmartNICs are capable of running full-fledged operating systems, but also ship with lightweight runtime systems that can provide kernel-bypass access to the NIC's IO engines.

Related work on SmartNIC offload. SmartNICs have been used to offload complex tasks from server CPUs. For example, StRoM [39] uses an FPGA NIC to implement RDMA verbs and creates generic kernels (or building blocks) that perform various functions, such as traversing linked lists. KV-Direct [30] uses an FPGA NIC to accelerate key-value accesses. iPipe [31] and Floem [36] are programming frameworks that simplify complex offload development for primarily CPU-based SmartNICs. E3 [32] transparently offloads microservices to SmartNICs.

The cost of SmartNICs. While SmartNICs provide the capabilities for complex offloads, they come at a cost. For example, a dual-port 25GbE BlueField SmartNIC at $2,340 costs $5.7\times$ more than the same-speed ConnectX-5 RNIC at $410 (cf. [13]). Another cost is the additional management required for SmartNICs. SmartNICs are a special piece of complex equipment that system administrators need to understand and maintain. SmartNIC operating systems and runtimes can crash, have security flaws, and need to be kept up-to-date with the latest vendor patches. This is an additional maintenance burden on operators that is not incurred by RNICs.

2.2 RDMA NICs

The processing power of RDMA NICs (RNICs) has doubled with each subsequent generation. This allows RNICs to cope with higher packet rates and more complex, hard-coded offloads (e.g., reduction operations, encryption, erasure coding).

We measure the verb processing bandwidth of several generations of Mellanox ConnectX NICs, using the Mellanox `ib_write_bw` benchmark. This benchmark performs 64B RDMA writes and, as such, it is not network bandwidth limited due to the small RDMA write size. We find that the verb processing bandwidth doubles with each generation, as we can
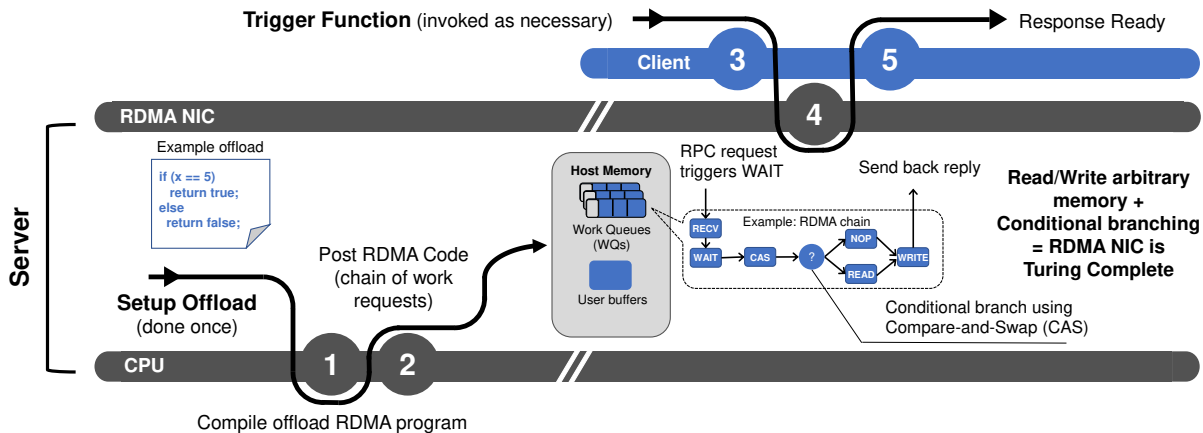
Figure 1: RDMA NICs can implement complex offloads if we allow conditional branches to be expressed. Conditional branching can be implemented by using CAS verbs to modify subsequent verbs in the chain, without any hardware modification.

see in Table 1. This is primarily due to a doubling in processing units (PUs) in each generation.[1] As a result, ConnectX-6 NICs can execute up to 110 million RDMA verbs per second using a single NIC port. This increased hardware performance further motivates the need for exploiting the computational power of these devices.

Related work on RDMA offload. RDMA has been employed in many different contexts, including accelerating key-value stores and filesystems [19, 22, 26, 35, 44], consensus [18, 27, 37, 41], distributed locking [45], and even nuanced use-cases such as efficient access in distributed tree-based indexing structures [46]. These systems operate within the confines of RDMA's intended use as a *data movement* offload (via remote memory access and message passing). When complex functionality is required, these systems involve multiple RDMA round-trips and/or rely on host CPUs to carry out the complex operations.

Within the storage context, Hyperloop [28] demonstrated that pushing the RNIC offload capabilities is possible. Hyperloop combines RDMA verbs to implement complex storage operations, such as chain replication, without CPU involvement. However, it does not provide a blueprint for offloading arbitrary processing and cannot offload functionality that uses any type of conditional logic (e.g., walking a remote data structure). Moreover, the Hyperloop protocol is likely incompatible with next-generation RNICs, as its implementation relies on changing work request ownership—a feature that is deprecated for ConnectX-4 and newer cards.

Unlike this body of previous work, we aim to unlock the general-purpose processing power of RNICs and provide an

unprecedented level of programmability for complex offloads, by using novel combinations of existing RDMA verbs (§3).

3 The RedN Computational Framework

To achieve our aforementioned goals, we develop a framework that enables complex offloads, called RedN. RedN's key idea is to combine widely available capabilities of RNICs to enable self-modifying RDMA programs. These programs—chains of RDMA operations—are capable of executing dynamic control flows with conditionals and loops. Fig. 1 illustrates the usage of RedN. The setup phase involves (①) preparing/compiling the RDMA code required for the service and (②) posting the output chain(s) of RDMA WRs to the RNIC. Clients can then use the offload by invoking a trigger (③) that causes the server's RNIC to (④) execute the posted RDMA program, which returns a response (⑤) to the client upon completion.

To further understand this proposed framework, we first look into the execution models offered by RNICs, and the ordering guarantees they provide for RDMA verbs. We then look into the expressivity of traditional RDMA verbs and explore parallels with CPU instruction sets. We use these insights to describe strategies for expressing complex logic using traditional RDMA verbs, *without requiring any hardware modifications.*

3.1 RDMA execution model

The RDMA interface specifies a number of data movement verbs (READ, WRITE, SEND, RECV, etc.) that are *posted* as *work requests* (WRs) by offload developers into *work queues* (WQs) in host memory. The RNIC starts execution of a sequence of WRs in a WQ once the offload developer triggers a *doorbell*—a special register in RNIC memory that informs the RNIC that a WQ has been updated and should be executed.

Work request ordering. Ordering rules for RDMA WRs distinguish between write WRs and non-write WRs that return a value. Within each category of operations, RDMA guarantees in-order execution of WRs within a single WQ. In particular, write WRs (i.e., SEND, WRITE, WRITEIMM) are totally or-

---

[1]Discussions with Mellanox affirmed our findings.

RNIC	PUs	Throughput
ConnectX-3 (2014)	2	15M verbs/s
ConnectX-5 (2016)	8	63M verbs/s
ConnectX-6 (2017)	16	112M verbs/s

Table 1: Number of Processing Units (PUs) and performance of various ConnectX generations.

(a) Completion order.
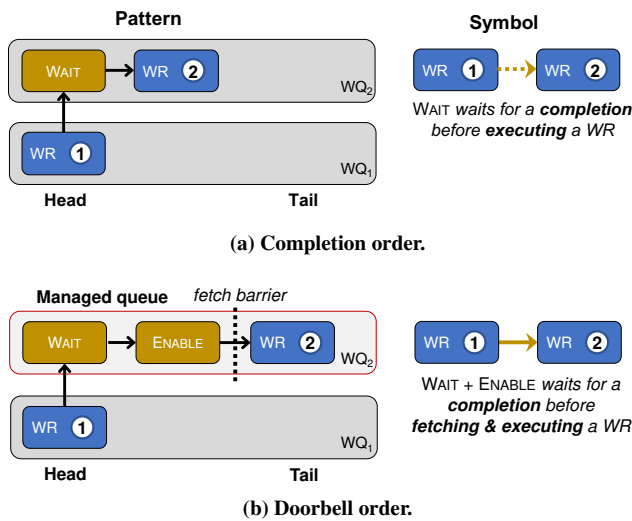
(b) Doorbell order.

Figure 2: Work request ordering modes that guarantee a total order of operations 2a and, a more restrictive "doorbell" order 2b, where operations are fetched by the NIC one-by-one. The symbols on the right will be used as notation for these WR chains in the examples of §3.

dered with regard to each other, but writes may be reordered before prior non-write WRs.

We call the default RDMA ordering mode *work queue (WQ) ordering*. Sophisticated offload logic often requires stronger ordering constraints, which we construct with the help of two RDMA verbs. Fig. 2 shows two stricter ordering modes that we introduce and how to achieve them.

The WAIT verb stops WR execution until the completion of a specified WR from another WQ or the preceding WR in the same WQ. We call this *completion ordering* (Fig. 2a). It achieves total ordering of WRs along the execution chain (which potentially involves multiple WQs). It can be used to enforce data consistency, similar to data memory barriers in CPU instruction sets—to wait for data to be available before executing the WRs operating on the data. Moreover, WAIT allows developers to *pre-post* chains of RDMA verbs to the RNIC, without immediately executing them.

In all the aforementioned ordering modes, the RNIC is free to prefetch into its cache the WRs within a WQ. Thus, the execution outcome reflects the WRs at the time they were fetched, which can be incoherent with the versions that reside in host memory in case these were later modified. To avoid this issue, the RNIC allows placing a WQ into *managed* mode, in which WR prefetch is disabled. The ENABLE verb is then used to explicitly start the prefetching of WRs. This allows for existing WRs to be modified within the WQ, as long as this is done before completion of the posted ENABLE—similar to an instruction barrier. We achieve a full (data and instruction) barrier, by using WAIT and ENABLE in sequence. We call this *doorbell ordering* (Fig. 2b). Doorbell ordering allows developers to modify WR chains in-place. In particular, it allows for *data-dependent, self-modifying* WRs.
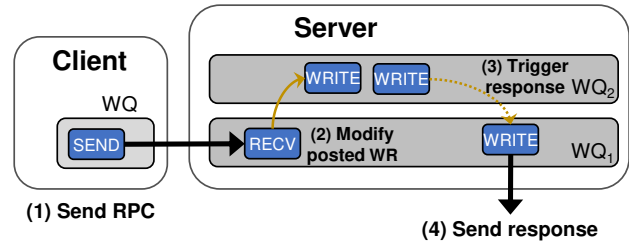
Figure 3: Clients can trigger posted operations. Thick solid lines represent (meta)data movements.

Thus, we have shown that we can control WR fetch and execution via special verbs, which we will exploit in the next section to develop full-fledged RDMA programs. These verbs are widely available in commodity RNICs (e.g., Mellanox terms them cross-channel communication [34]).

3.2 Dynamic RDMA Programs

While a static sequence of RDMA WRs is already a rudimentary RDMA program, complex offloads require *data-dependent execution*, where the logic of the offload is dependent on input arguments. To realize data-dependent execution, we construct *self-modifying RDMA code*.

Self-modifying RDMA code. Doorbell ordering enables a restricted form of self-modifying code, capable of data-dependent execution. To illustrate this concept, we use the example of a server host that offloads an RPC handler to its RNIC as shown in Fig. 3. The RPC response depends on the argument set by the client and thus the RDMA offload is data-dependent. The server posts the RDMA program that consists of a set of WRs spanning two WQs. The client invokes the offload by issuing a SEND operation. At the RNIC, the SEND triggers the posted RECV operation. Observe that RECV specifies where the SEND data is placed. We configure RECV to inject the received data into the posted WR chain in WQ_2 to modify its attributes. We achieve this by leveraging doorbell ordering, to ensure that posted WRs are not prefetched by the RNIC and can be altered by preceding WRs.

This is an instance of self-modifying code. As such, clients can pass arguments to the offloaded RPC handler and the RNIC will dynamically alter the executed code accordingly. However, this by itself is not sufficient to provide a Turing complete offload framework.

Turing completeness of RDMA. Turing completeness implies that a system of data-manipulation rules, such as RDMA, are computationally universal. For RDMA to be Turing complete, we need to satisfy two requirements [25]:
T1: Ability to read/write arbitrary amounts of memory.
T2: Conditional branching (e.g. if/else statements).

T1 can be satisfied for limited amounts of memory with regular RDMA verbs, whereas T2 has not been demonstrated with RDMA NICs. However, to truly be capable of accessing an *arbitrary* amount of memory, we need a way of realizing loops. Loops open up a range of sophisticated use-cases and

Pseudocode
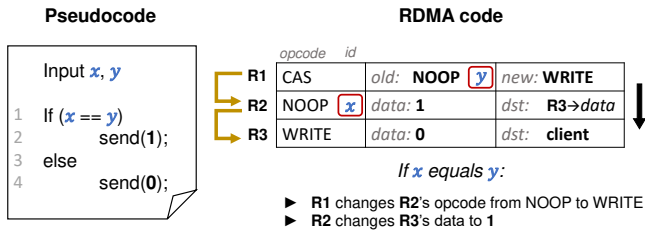
RDMA code

Figure 4: Simple *if* example and equivalent RDMA code. Conditional execution relies on self-modifying code using CAS to enable/disable WRs based on the operand values.

lower the number of constraints that programmers have to consider for offloads. To highlight their importance, we add them as a third requirement, necessary to fulfill the first:
T3: The ability to execute code repeatedly (loops).

In the next sub-sections, we show how dynamic execution can be used to satisfy all the aforementioned requirements. A proof sketch of Turing completeness is given in Appendix A.

3.3 Conditionals

Conditional execution—choosing what computation to perform based on a runtime condition—is typically realized using conditional branches, which are not readily available in RDMA. To this end, we introduce a novel approach that uses self-modifying CAS verbs. The main insight is that this verb can be used to check a condition (i.e., equality of x and y) and then perform a swap to modify the attributes of a WR. We describe how this is done in Fig. 4. We insert a CAS that compares the 64-bit value at the address of R2's *opcode* attribute (initially NOOP) with its *old* parameter (also initially NOOP). We then set the *id* field of R2 to x. This field can be manipulated freely without changing the behavior of the WR, allowing us to use it to store x. Operand y is stored in the corresponding position in the *old* field of R1. This means that if x and y are equal, the CAS operation will succeed and the value in R1's *new* field—which we set to WRITE—will replace R2's opcode. Hence, in the case $x = y$, R2 will change from a NOOP into a WRITE operation. This WRITE is set to modify the *data* value of the return operation (R3) to 1. If x and y are not equal, the default value 0 is returned.

Now that we have established the utility of this technique for basic conditionals, we next look into how to can be used to support loop constructs.

3.4 Loops

To support loop constructs efficiently, we require (1) conditional branching to test the loop condition and break if necessary, and (2) WR re-execution, to repeat the loop body. We develop each, in turn, below.

Consider the while loop example in Figure 5. This offload searches for x in an array A and sends the corresponding index. The loop is static because A has finite size (in this case, size =2), known a priori. To simplify presentation, consider the case $A[i] = i, \forall i$. Without this simplification, the example would include an additional WRITE to fetch the value at $A[i]$.
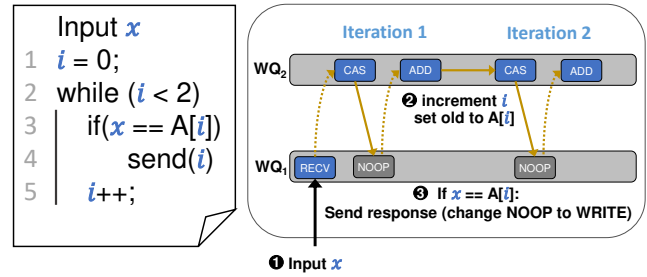
Figure 5: while loop using CAS. Loop is unrolled since loop size is fixed and set to 2.
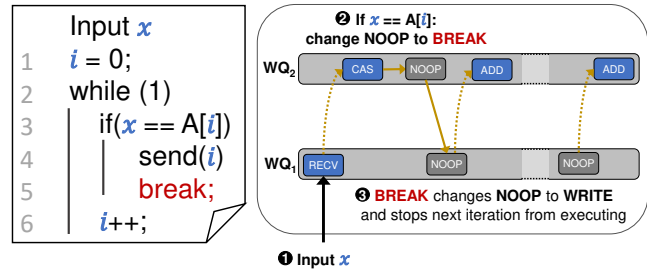
Figure 6: *while* loop with breaks realized using CAS. To implement breaks, we use CAS to change a NOOP WR to an RDMA WRITE, which then stops subsequent iterations from executing.

The loop body uses a CAS verb to implement the if condition (line 3), followed by an ADD verb to increment i (line 6). Given that the loop size is known a priori (size = 2), RedN can unroll the while loop in advance and post the WRs for all iterations. As such, there is no need to check the condition at line 2. For each iteration, if the CAS succeeds, the NOOP verb in WQ_1 will be changed to WRITE—which will send the response back to the client. However, it is clear that, regardless of the comparison result, all subsequent iterations will be executed. This is inefficient since, if the send (line 4) occurs before the loop is finished, a number of WRs will be wastefully executed by the NIC. This is impractical for larger loop sizes or if the number of iterations is not known a priori.

Unbounded loops and termination. Figure 6 modifies the previous example to make it such that the loop is unbounded. For efficiency, we add a break that exits the loop if the element is found. The role of break is to prevent additional iterations from being executed. We use an additional NOOP that is formatted such that, once transformed into a WRITE by the CAS operation, it prevents the execution of subsequent iterations in the loop. This is done by modifying the last WR in the loop such that it does not trigger a completion event. The next iteration in the loop, which WAITs on such an event (via completion ordering), will therefore not be executed. Moreover, the WRITE will also modify the opcode of the WR used to send back the response from NOOP to WRITE.

As such, break allows efficient and unbounded loop execution. However, it still remains necessary for the CPU to post WRs to continue the loop after all its WRs are executed. This consumes CPU cycles and can even increase latency if the CPU is unable to keep up with the speed of WR execution.

RedN Constructs		Number of WRs	Operand limit [bits]
if		1C + 1A + 3E	
while	Unrolled	1C + 1A + 3E	48
	Recycled	3C + 2A + 4E	

Table 2: Breakdown of the overhead of our constructs with different offload strategies. *C* refers to copy verbs, *A* refers to atomic RDMA verbs, and *E* refers to WAIT/ENABLE verbs. while loops that use WQ recycling incur 2 additional READs, 1 ADD, and 1 ENABLE WR.

Unbounded loops via WQ recycling. To allow the NIC to recycle WRs without CPU intervention, we make use of a novel technique that we call *WQ recycling*. RNICs iterate over WQs, which are circular buffers, and execute the WRs therein. By design, each WR is meant to execute only once. However, there is no fundamental reason why WRs cannot be reused since the RNIC does not actually erase them from the WQ. To enable recycling of a WR chain, we insert a WAIT and ENABLE sequence at the tail of the WQ. This instructs the RNIC to wrap around the tail and re-execute the WR chain for as many times as needed.

It is important to note that WQ recycling is not a panacea. To allow the tail of the WQ to wrap around, all posted WAIT and ENABLE WRs in the loop need to have their *wqe_count* attribute updated. This attribute is used to determine the index of the WR that these ordering verbs affect. In ConnectX NICs, these indices are maintained internally by the RNIC and their values are monotonically increasing (instead of resetting after the WQ wraps around). As such, the *wqe_count* values need to be incremented to match. This incurs overhead (as seen in Table 2) and requires an additional ADD operation in combination with other verbs. As such, loop unrolling, where each iteration is manually posted by the CPU, is overall less taxing on the RNIC. However, WQ recycling avoids CPU intervention, allowing the offload to remain available even amid host software failures (as we will see later in §5.6).

3.5 Putting it all together

With conditional branching, we can dynamically alter the control flow of any function on an RNIC. Loops allow us to traverse arbitrary data structures. Together, we have transformed an RNIC into a general processing unit. In this section, we discuss the usability aspects from overhead, security, programmability, and expressiveness perspectives.

Building blocks. We abstract and parameterize the RDMA chains required for conditional branching and looping into if and while constructs. The overhead in terms of RDMA WR chains of our constructs is shown in Table 2. We can see a breakdown of the minimum number of operations required for each. Inequality predicates, such as < or >, can also be supported by combining equality checks with MAX or MIN, as seen later in Table 3. However, their availability is vendor-specific and currently only supported by ConnectX NICs.

Operand limits. RedN's limit is based on the supported size for the CAS verb, which is 64 bits. The operand is provided

as a 48-bit value, encoded in its *id* and other neighboring fields (which can also be freely modified without affecting execution). The remaining bits are used for modifying the opcode of the WR depending on the result of the comparison. We note that our advertised limits only signify what is possible with the number of operations we allocate for our constructs. For instance, despite the 48-bit operand limit for our constructs, we can chain together multiple CAS operations to handle different segments of a larger operand (we do not rely on the atomicity property of CAS). As such, there is no fundamental limitation, only a performance penalty.

Offload setup. To offload an RDMA program, clients first create an RDMA connection to the target server and send an RPC to initiate the offload. We envision that the server already has the offload code; however, other ways of deploying the offload are possible. Upon receiving a connection request, the server creates one or more managed local WQs to post the offloaded code. Next, it registers two main types of memory regions for RDMA access: (a) a code region, and (b) a data region. The code region is the set of remote RDMA WQs created on the server, which are unique to each client and need to be accessible via RDMA to allow self-modifying code. Code regions are protected by memory keys—special tokens required for RDMA access—upon registration (at connection time), prohibiting unauthorized access. The data region holds any data elements used by the offload (e.g., a hash table). Data regions can be shared or private, depending on the use-case.

Security. RedN does not solve security challenges in existing RDMA or Infiniband implementations [40]. However, RedN can help RDMA systems become more secure. For such systems, *one-sided* RDMA operations (e.g., RDMA READ and WRITE) are frequently used [22,28,33,35,42,43] as they avoid CPU overheads at the responder. However, doing so requires clients to have direct read and/or write memory access. This can compromise security if clients are buggy and/or malicious. To give an example, FaRM allows clients to write messages directly to shared RPC buffers. This requires clients to behave correctly, as they could otherwise overwrite or modify other clients' RPCs. RedN allows applications to use *two-sided* RDMA operations (e.g., SEND and RECV), which do not require direct memory access, while *still* fully bypassing server CPUs. As we demonstrate in our use-cases in §5, SEND operations can be used to trigger offload programs without any CPU involvement.

Isolation. Given that RedN implements dynamic loops, clients can abuse such constructs to consume more than their fair share of resources. Luckily, popular RNICs, like ConnectX, provide WQ rate-limiters [6] for performance isolation. As such, even if clients trigger non-terminating offload code, they still have to adhere to their assigned rates. Moreover, offloaded code can be configured by the servers to be auditable through completion events, created automatically after a WR is executed. These events can be monitored and servers can terminate connections to clients running misbehaving code.

Parallelism. RDMA WR fetch and execution latencies are more costly compared to CPU instructions, as WRs are fetched/executed via PCIe (microseconds vs. nanoseconds). As such, to hide WR latencies, it is important to parallelize logically unrelated operations. Like threads of execution in a CPU, each WQ is allocated a single RNIC PU to ensure in-order execution without inter-PU synchronization. As such, we carefully tune our offloaded code to allow unrelated verbs to execute on independent queues to be able to parallelize execution as much as possible. The benefits of parallelism are evaluated in §5.2.

4 Implementation

Our offload framework is implemented in C with ~2,300 lines of code—this includes our use cases (~1400), and convenience wrappers for RDMA verbs (*libibverbs*) API (~900).

Our approach does not require modifying any RDMA libraries or drivers. RedN uses low-level functions provided by Mellanox's ConnectX driver (*libmlx5*) to expose in-memory WQ buffers and register them to the RNIC, allowing WRs to be manipulated via RDMA verbs. We configure the ConnectX-5 firmware to allow the WR *id* field to be manipulated freely, which is required for conditional operations as well as WR recycling. This is done by modifying specific configuration registers on the NIC [12].

RedN is compatible with any ConnectX NICs that support WAIT and ENABLE (e.g., ConnectX-3 and later models).

5 Evaluation

We start by characterizing the underlying RNIC performance (§5.1) to understand how it affects our implemented programming constructs. Then, in our evaluation against state-of-the-art RNIC and SmartNIC offloads, we show that RedN:

1. Speeds up remote data structure traversals, such as hash tables (§5.2) and linked lists (§5.3) compared to vanilla RDMA offload;
2. Accelerates (§5.4) and provides performance isolation (§5.5) for the Memcached key-value store;
3. Provides improved availability for applications (§5.6)—allowing them to run in spite of OS & process crashes;
4. Exposes programming constructs generic enough to enable a wide-variety of use-cases (§5.2–§5.6);

Testbed. Our experimental testbed consists of 3× dual-socket Haswell servers running at 3.2 GHz, with a total of 16 cores, 128 GB of DRAM, and 100 Gbps dual-port Mellanox ConnectX-5 Infiniband RNICs. All nodes are running Ubuntu 18.04 with Linux Kernel version 4.15 and are connected via back-to-back Infiniband links.

NIC setup. For all of our experiments, we use reliable connection (RC) RDMA transport, which supports the RDMA synchronization features we use. All WQs that enforce doorbell order are initialized with a special "managed" flag to disable the driver from issuing doorbells after a WR is posted. The WQ size is set to match that of the offloaded program.
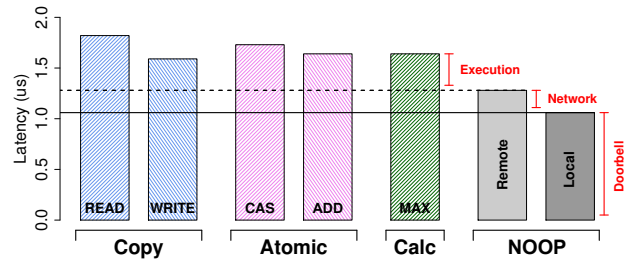
Figure 7: Latencies of different RDMA verbs. The solid line marks the latency of ringing the doorbell via MMIO. The difference between dashed and solid lines estimates network latency.

5.1 Microbenchmarks

We run microbenchmarks to break down RNIC verb execution latency, understand the overheads of our different ordering modes, and determine the processing bandwidth of different RDMA verbs and of our constructs.

5.1.1 RDMA Latency

We break down the performance of RDMA verbs, configured to perform 64B IO, by measuring their average latencies after executing them 100K times. All verbs are executed remotely, unless otherwise stated. As seen in Fig. 7, WRITE has a latency of 1.6 μs. It uses posted PCIe transactions, which are one-way. Comparatively, non-posted verbs such as READ or atomics such as fetch-and-add (ADD) and compare-and-swap (CAS) need to wait for a PCIe completion and take ~1.8 μs.[2] Overall, the execution time difference is small among verbs, even for more advanced, vendor-specific *Calc* verbs that perform logical and arithmetic computations (e.g., MAX).

To break down the different latency components for RDMA verb execution, we first estimate the latency of issuing a doorbell and copying the WR to the RNIC. This can be done by measuring the execution time of a NOOP WR. This time can be subtracted from the latencies of other WRs to give an estimate of their execution time once the WR is available in the RNIC's cache. We also quantify the network cost by executing remote and local loopback NOOP WRs (shown on the right-hand side) and measuring the difference—roughly 0.25 μs for our back-to-back connected nodes. Overall, these results show low verb execution latency, justifying building more sophisticated functions atop. We next measure the implications of ordering for offloads.

5.1.2 Ordering Overheads

We show the latency of executing chains of RDMA verbs using different ordering modes. All the posted WRs within a chain are NOOP, to simplify isolating the performance impact of ordering. We start by measuring the latency of executing a chain of verbs posted to the same queue but absent any constraints (WQ order), and compare it to the ordering modes

---

[2]Older-generation NICs (e.g., ConnectX-4) use a proprietary concurrency control mechanism to implement atomics, resulting in higher latencies than later generations that rely on PCIe atomic transactions.
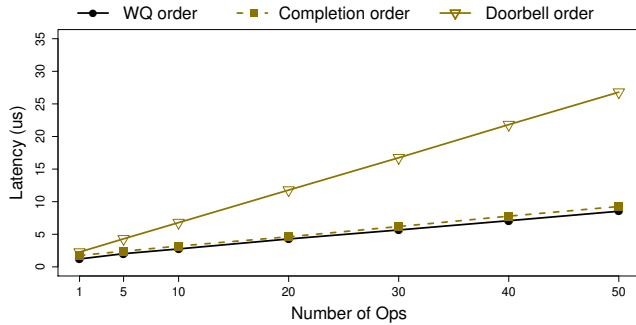
Figure 8: Execution latency of RDMA verbs posted using different ordering modes. More restrictive modes such as Doorbell order add non-negligible overheads as it requires the NIC to fetch WRs sequentially.

that we introduced in Fig. 2—completion order and doorbell order. WQ order only mandates in-order updates to memory, which allows for increased concurrency. Operations that are not modifying the same memory address can execute concurrently and the RNIC is free to prefetch multiple WRs with a single DMA[3]. We can see in Fig. 8 that the latency of a single NOOP is 1.21 µs and the overhead of adding subsequent verbs is roughly 0.17 µs per verb. The first verb is slower since it requires an initial doorbell to tell the NIC that there is outstanding work. For completion ordering, less concurrency is possible since WRs await the completions of their predecessors, and the overhead of increases slightly to 0.19 µs per additional WR. For doorbell order, no latency-hiding is possible, as the NIC has to fetch WRs from memory one-by-one, which results in an overhead of 0.54 µs per additional WR. These results signify that, doorbell ordering should be used conservatively, as there is more than 0.5 µs latency increase for every instance of its use, compared to more relaxed ordering modes.

5.1.3 RDMA Verb Throughput

We show the throughput of the common RDMA verbs in Table 3 for a single ConnectX-5 port. ConnectX cards assign compute resources on a per port basis. For ConnectX-5, each port has 8 PUs. Atomic verbs, such as CAS, offer a comparatively limited throughput (8× lower than regular verbs) due to memory synchronization across PCIe.

In addition, we measure the performance of RedN's if and while constructs. Using 48-bit operands, a ConnectX-5 NIC can execute 700K if constructs per second. This is due to the need for CAS to ensure doorbell ordering between CAS and the subsequent WR it modifies. This causes the throughput to be bound by NIC processing limits. Unrolled while loops require the same number of verbs per iteration as an if statement and their throughput is identical. while loops with WQ recycling have reduced performance due to having to execute more WRs per iteration.

---

[3]The number of operations fetched by the RNIC can change dynamically. The Prefetch mechanism in ConnectX RNICs is proprietary.

Operation		Throughput (M ops/s)	Support
Atomic	CAS	8.4	Native
	ADD		
Copy	READ	65	
	WRITE	63	
Calc	MAX	63	Mellanox
Constructs	if	0.7	RedN
	while Unrolled	0.7	
	Recycled	0.3	

Table 3: Throughput of common RDMA verbs and RedN's constructs on a single port of a ConnectX-5. if and unrolled while have identical performance. while loops with WQ recycling require additional WRs and therefore have a lower throughput.

5.2 Offload: Hash Lookup

After evaluating the overheads of RedN's ordering modes and constructs, we next look into the performance of RedN for offloading remote access to popular data structures. We first look into hash tables, given their prominent use in key-value stores for indexing stored objects. To perform a simple *get* operation, clients first have to lookup the desired key-value entry in the hash table. The entry can either have the value directly inlined or a pointer to its memory address. The value is then fetched and returned back to the client. Hopscotch hashing is a popular hashing scheme that resolves collisions by using *H* hashes for each entry and storing them in 1 out of *H* buckets. Each bucket has a *neighborhood* that can probabilistically hold a given key. A lookup might require searching more than one bucket before the matching key-value entry is found. To support dynamic value sizes, we assume the value is not inlined in the bucket and is instead referenced via a pointer.

For distributed key-value stores built with RDMA, *get* operations are usually implemented in one of two ways:

One-sided approaches first retrieve the key's location using a one-sided RDMA READ operation and then issue a second READ to fetch the value. These approaches typically require two network round-trips at a minimum. This greatly increases latency but does not require involvement of the server's CPU. Many systems utilize this approach to implement lookups, including FaRM [22] and Pilaf [35].

Two-sided approaches require the client to send a request using an RDMA SEND or WRITE. The server intercepts the request, locates the value and then returns it using one of the aforementioned verbs. This widely used [19, 26] approach follows traditional RPC implementations and avoids the need for several roundtrips. However, this comes at the cost of server CPU cycles.

5.2.1 RedN's Approach

To offload key-value *get* operations, we leverage the offload schemes introduced in §3.3 and §3.4.

Fig. 9 describes the RDMA operations involved for a single-hash lookup. To *get* a value corresponding to a key, the client first computes the hashes for its key. For this use-case, we set the number of hashes to two, which is common in practice [24]. The client then performs a SEND with the value of
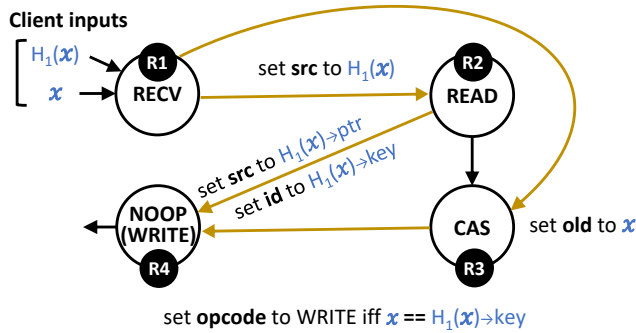
Figure 9: Hash lookup RDMA program. Black arrows indicate order of execution of WRs in their WQs. Brown arrows indicate self-modifying code dependencies and require doorbell ordering. x is the requested key and $H_1(x)$ is its first hash. The acronym *src* indicates the "source address" field of WRs. *old* indicates the "expected value" at the target address of the CAS operation. The *id* field is used for storing conditional operands.

the key x and address of the first bucket $H_1(x)$, which are then captured via a RECV WR posted on the server. The RECV WR (R1) inserts x into the *old* field of the CAS WR (R3) and the bucket address $H_1(x)$ into the READ WR (R2). The READ WR retrieves the bucket and sets the source address (*src*) of the response WR (R4) to the address of the value (*ptr*). It also inserts the bucket's key into the *id* field to prepare it for the conditional check. Finally, CAS (R3) checks whether the expected value *old*, which is set to key x, matches the *id* field in (R4), which is set to the bucket's *key*. If equal, (R4)'s opcode is changed from NOOP to WRITE, which then returns the value from the bucket. Given that each key may be stored in multiple buckets (two in our setup), these lookups may be performed sequentially or in parallel, depending on the offload configuration.

5.2.2 Results

We evaluate our approach against both one-sided and two-sided implementations of key-value *get* operations. We use FaRM's approach [22] to perform one-sided lookups. FaRM uses Hopscotch hashing to locate the key using approximately two RDMA READs — one for fetching the buckets in a neighborhood that hold the key-value pairs and another for reading the actual value. The neighborhood size is set to 6 by default, implying a $6\times$ overhead for RDMA metadata operations. For
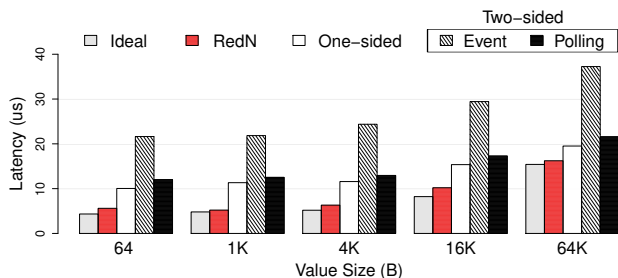
Figure 10: Average latency of hash lookups. *Ideal* shows the latency of a single network round-trip READ.
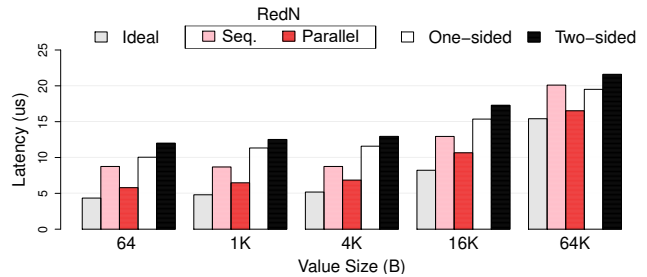
Figure 11: Average latency of hash lookups during collisions. *Ideal* shows the latency of a single network round-trip READ.

two-sided lookups, our RPC to the host involves a client-initiated RDMA SEND to transmit the *get* request, and an RDMA WRITE initiated by the server to return the value after performing the lookup.

Latency. Fig. 10 shows a latency comparison of KV *get* operations of RedN against one-sided and two-sided baselines. We evaluate two distinct variations of two-sided. The *event-based* approach blocks for a completion event to avoid wasting CPU cycles, whereas the *polling-based* approach dedicates one CPU core for polling the completion queue. We use 48-bit keys and vary the value size. The value size is given on the x-axis. In this scenario, we assume no hash collisions and that all keys are found in the first bucket. RedN is able to outperform all baselines — fetching a 64 KB key-value pair in 16.22 μs, which is within 5% of a single network round-trip READ (Ideal). RedN is able to deliver close-to-ideal performance because it bypasses the server's CPU **and** fetches the value in a single network RTT. Compared to RedN, one-sided operations incur up to $2\times$ higher latencies, as they require two RTTs to fetch a value. Two-sided implementations do not incur any extra RTT; however, they require server CPU intervention. The polling-based variant consumes an entire CPU core but provides competitive latencies. Event-based approaches block for completion events to avoid wasting CPU cycles and incur much higher latencies as a consequence. RedN is able to outperform polling-based and event-based approaches by up to 2 and $3.8\times$, respectively. Given the much higher latencies of event-based approaches, for the remainder of this evaluation, we will only focus on polling-based approaches and simply refer to them hereafter as *two-sided*.

Fig. 11 shows the latency in the presence of hash collisions. In this case, we assume a worst case scenario, where the key-value pair is always found in the second bucket. In this scenario, we introduce two offload variants for RedN— RedN-Seq & RedN-Parallel. The former performs bucket lookups sequentially within a single WQ. The latter parallelizes bucket lookups by performing the lookups across two different WQs to allow execution on different NIC PUs. We can see that RedN-Parallel maintains similar latencies to lookups with no hash collisions (i.e., *RedN* in Fig. 10), since bucket lookups are almost completely parallelized. It is worth noting that parallelism in this case does not cause unnecessary data movement, since the value is only returned when the corresponding

Hash lookup	IO Size			
	≤ 1 KB		64 KB	
	Single	Dual	Single	Dual
Port config.	Single	Dual	Single	Dual
Rate (ops/s)	500K	1M	180K	190K
Bottleneck	NIC PU		IB bw	PCIe bw

Table 4: NIC throughput of hash lookups and its bottlenecks.
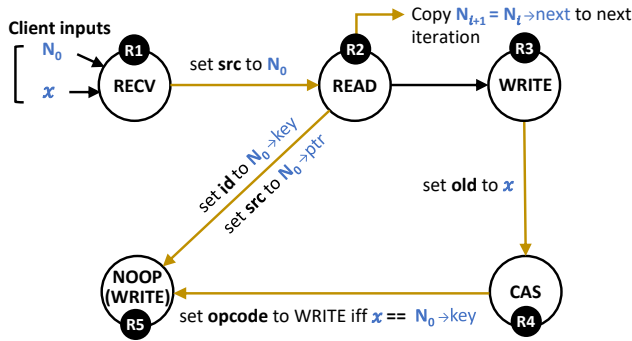
Figure 12: Linked list RDMA program.

key is found. For the other bucket, the WRITE operation (R4 in Fig. 9) is a NOOP. RedN-Seq, on the other hand, incurs at least 3 μs of extra latency as it needs to search the buckets one-by-one. As such, whenever possible, operations with no dependencies should be executed in parallel. The trade-off is having to allocate extra WQs for each level of parallelism.

Throughput. We describe our throughput in Table 4. At lower IO, RedN is bottlenecked by the NIC's processing capacity due to the use of doorbell ordering—reaching 500K ops/s on a single port (1M ops/s with dual ports). At 64 KB, RedN reaches the single-port IB bandwidth limit (~ 92 Gbps). Dual-port configs are limited by ConnectX-5's 16× PCIe 3.0 lanes.

SmartNIC comparison. We compare our performance for hashtable *gets* against StRoM [39], a programmable FPGA-based SmartNIC. Since we do not have access to a programmable FPGA, we extract the results from [39] for comparison, and report them in Table 5. RedN uses the same experimental settings as before. Our hashtable configuration is functionally identical to StRoM's and our client and server nodes are also connected via back-to-back links. We can see that RedN provides lower lookup latencies than StRoM. StRoM uses a Xilinx Virtex 7 FPGA, which runs at 156.25 MHz, and incurs at least two PCIe roundtrips to retrieve the key and value. Our evaluation shows that RedN can provide latency that is in-line with more expensive SmartNICs.

IO Size	System	Median	99thile

IO Size	System	Median	99^{th}ile
64 B	RedN	5.7 μs	6.9 μs
	StRoM	~7 μs	~7 μs
4 KB	RedN	6.7 μs	8.4 μs
	StRoM	~12 μs	~13 μs

Table 5: Latency comparison of hash *gets*. Results for StRoM obtained from [39].
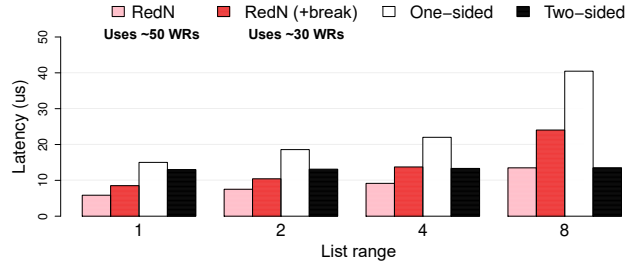
Figure 13: Average latency of walking linked lists.

5.3 Offload: List Traversal

Next, we explore another data structure also popularly used in storage systems. We focus on linked lists that store key-value pairs, and evaluate the overhead of traversing them remotely using our offloads. Similar to the previous use-case, we focus on one-sided approaches, as used by FaRM and Pilaf [22, 35].

Linked list processing can be decomposed into a *while* loop for traversing the list and an *if* condition for finding and returning the key. We describe the implementation of our offload in Fig. 12. The client provides the key *x* and address of the first node in the list N_0. A READ operation (R2) is then performed to read the contents of the first node and update the values for the return operation (R5). We also use a WRITE operation (R3) to prepare the CAS operation (R4) by inserting key *x* in its *old* field. As an optimization, this WRITE can be removed and, instead, *x* can be inserted directly by the RECV operation. This, however, will need to be done for every CAS to be executed and, as such, this approach is limited to smaller list sizes, since RECVs can only perform 16 scatters.

For this use-case, we introduce two offload variations. The first, referred to simply as RedN, uses the implementation in Fig. 12. The second uses an additional *break* statement between R4 and R5 to exit the loop in order to avoid executing any additional operations.

5.3.1 Results

Fig. 13 shows the latency of one-sided and two-sided variants against RedN at various linked list ranges — where range represents the highest list element that the key can be randomly placed in. The size of the list itself is set to a constant value of 8. We setup the linked list to use key and value sizes of 48 bits and 64 bytes, respectively, and perform 100k list traversals for each system. The requested key is chosen at random for each RPC. In the variant labelled "RedN", we do not use *breaks* and assume that all 8 elements of the list need to be searched. RedN outperforms all baselines for all list ranges until 8 — providing up to a 2× improvement. *RedN (+break)* executes a break statement with each iteration and performs worse than RedN due to the extra overhead of checking the condition of the *break*. However, using a break statement increases the offload's overall efficiency since no unneeded iterations are executed after the key is found — using an average of 30 WRs across all experiments. Without breaks, RedN will need to execute all subsequent iterations even after the key-value
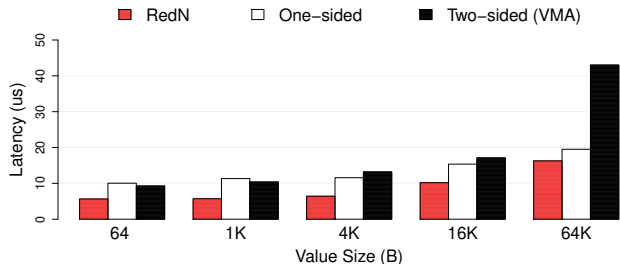
Figure 14: Memcached *get* latencies with different IO sizes.

Figure 15: Memcached *get* latencies under hardware contention with varying numbers of writer-clients.

pair is found/returned and it uses more than 65% more WRs. As such, while RedN is able to provide better latencies, using a break statement is more sensible for longer lists.

5.4 Use Case: Accelerating Memcached

Based on our earlier experience offloading remote data structure traversals, we set out to see: 1) how effective our aforementioned techniques are in a real system, and 2) what are the challenges in deploying it in such settings. Memcached is a key-value store that is often used as a caching service for large-scale storage services. We use a version of Memcached that employs cuckoo hashing [24]. Since Memcached does not natively support RDMA, we modify it with ~700 LoC to integrate RDMA capabilities, allowing the RNIC to register the hash table and storage object memory areas. We also modify the buckets, so that the addresses to the values are stored in big endian — to match the format used by the WR attributes. We then use RedN to offload Memcached's *get* requests to allow them to be serviced directly by the RNIC without CPU involvement. We compare our results to various configurations of Memcached.

To benchmark Memcached, we use the Memtier benchmark, configure it to use UDP (to reduce TCP overheads for the baselines), and issue 1 million *get* operations using different key-value sizes. To create a competitive baseline for two-sided approaches, we use Mellanox's VMA [9]—a kernel-bypass userspace TCP/IP stack that boosts the performance of sockets-based applications by intercepting their socket calls and using kernel-bypass to send/receive data. We configure VMA in polling-mode to optimize for latency. In addition, we also implement a one-sided approach, similar to the one introduced in section 5.2.

Fig. 14 shows the latency of *gets*. As we can see, RedN's offload for hash *gets* is up to $1.7\times$ faster than one-sided and $2.6\times$ faster than two-sided. Despite the latter being configured in polling-mode, VMA incurs extra overhead since it relies on a network stack to process packets. In addition, to adhere to the sockets API, VMA has to memcpy data from send and receive buffers, further inflating latencies—which is why it performs comparatively worse at higher value sizes.

5.5 Use Case: Performance Isolation

One of the benefits of exposing the latent turing power of RNICs is to enforce isolation among applications. CPU con-

tention in multi-tenant and cloud settings can lead to arbitrary context switches, which can, in turn, inflate average and tail latencies. We explore such a scenario by sending background traffic to Memcached using one or more writer (clients). These writers generate *set* RPCs in a closed loop to load the Memcached service. At the same time, we use a single reader client to generate *get* operations. To stress CPU resources while minimizing lock contention, each reader/writer is assigned a distinct set of 10K keys, which they use to generate their queries. The keys within each set are accessed by the clients sequentially.
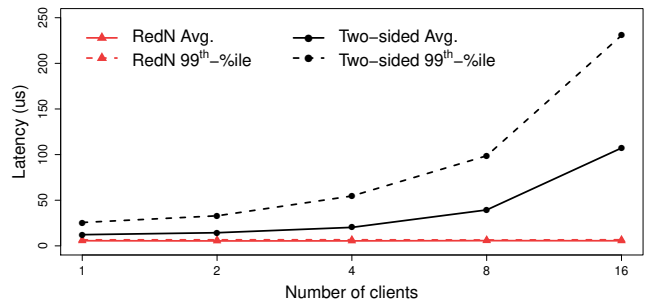
We can see in Fig. 15 that, as we increase the # of writers, both the average and 99th percentile latencies for two-sided increase dramatically. For RedN, CPU contention has no impact on the performance of the RNIC and both the average and 99th percentiles sit below 7 µs. At 16 writers, RedN's 99th percentile latency is $35\times$ lower than the baseline.

This indicates that RNIC offloads can also have other useful effects. Service providers may opt to offload high priority traffic for more predictable performance or allocate server resources to tenants to reduce contention.

5.6 Use Case: Failure Resiliency

We now consider server failures and how failure is affected by RNICs. Table 6 shows failure rates of server software and hardware components. NICs are much less likely to fail than software components—NIC annualized failure rate (AFR) is an order of magnitude lower. Even more importantly, NICs are partially decoupled from their hosts and can still access memory (or NVM) in the presence of an OS failure. This means that RNICs are capable of offloading key system functionality that can allow servers to continue operating despite OS failures (albeit in a degraded state). To put this to the test, we conduct a fail-over experiment to explore how RedN can enhance a service's failure resiliency.

Process crashes. We look into how we can allow an RNIC to continue serving RPCs after a Memcached instance crashes. We find that this is not simple in practice. RNICs access many resources in application memory (e.g., queues, doorbell records, etc.) that are required for functionality. If the process hosting these resources crashes, the memory belonging to

Component	AFR	MTTF	Reliability
OS	41.9%	20,906	99%
DRAM	39.5%	22,177	99%
NIC	1.00%	876,000	99.99%
NVM	< 1.00%	2 million	99.99%

Table 6: Failure rates of different server components [8, 37]. AFR means annualized failure rate, whereas MTTF stands for mean time to failure and is expressed in hours. RNICs can still access memory even in the presence of an OS failure.
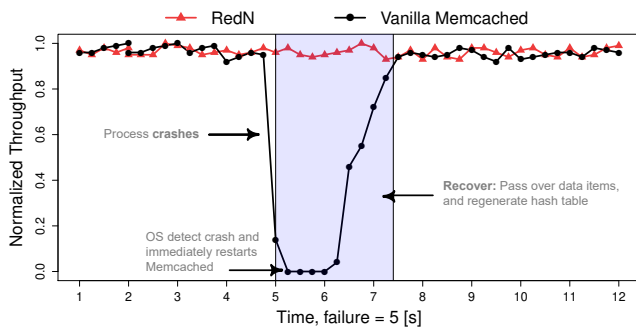
Figure 16: RedN can survive process crashes and continue serving RPCs via the RNIC without interruption.

these components will be automatically freed by the operating system resulting in termination of the RDMA program. To counteract this, we use [38] forks to create an empty hull parent for hosting RDMA resources and then allow Memcached to run as a child process. Linux systems do not free the resources of a crashed child until the parent also terminates. As such, keeping the RDMA resources tied to an empty process allows us to continue operating in spite of application failures. We run an experiment (timeline shown in Fig. 16) where we send *get* queries to a single instance of Memcached and then simply kill Memcached during the run. The OS detects the application's termination and immediately restarts it. Despite this, we can see that a vanilla Memcached instance will take at least 1 second to bootstrap, and 1.25 additional seconds to build its metadata and hashtables. With RedN, no service disruption is experienced and *get* queries continue to be issued without recovery time.

OS failure. We also programmatically induce a kernel panic using sysctl, freezing the system. This is a simpler case than process crashes, since we no longer have to worry about the OS freeing RDMA resources. For brevity, we do not show these results, but we experimentally verified that RedN offloads continue operating in the presence of an OS crash.

6 Discussion

Client scalability. RedN requires servers to manage at least two WQs per client, which is not higher than other RDMA systems. RedN can still introduce scalability challenges with thousands of clients since RNIC cache is limited. However, Mellanox's dynamically-connected (DC) transport service [5], which allows unused connections to be recycled, can circumvent many such scalability limits.

Offload for sockets-based applications. Protocols such as rsocket [16] can be used to transparently convert sockets-based applications to use RDMA, making them possible targets for RedN. Although rsocket does not support popular system calls, such as epoll, other extensions have been proposed [29] that support a more comprehensive list of system calls and were shown to work with applications like Memcached and Redis.

Intel RNICs. Next-generation Intel RNICs are expected to support atomic verbs, such as CAS—which RedN uses to implement conditionals. To control when WRs can be fetched by the NIC, Intel uses a validity bit in each WR header. This bit can be dynamically modified via an RDMA operation to mimic ENABLE. However, there is no equivalent for the WAIT primitive, meaning that clients cannot trigger a pre-posted chain. One possible workaround for this is to use another PCIe device on the server to issue a doorbell to the RNIC, allowing the WR chain to be triggered. We leave the exploration of such techniques as future work.

Insights for next-generation RNICs. Our experience with RedN has shown that keeping WRs in server memory (to allow them to be modified by other RDMA verbs) is a key bottleneck. If the NIC's cache was made directly accessible via RDMA, WRs can be pre-fetched in advance and unnecessary PCIe round-trips on the critical path can be avoided. We hope future RNICs will support such features.

7 Conclusion

We show that, in spite of appearances, commodity RDMA NICs are Turing-complete and capable of performing complex offloads without *any* hardware modifications. We take this insight and explore the feasibility and performance of these offloads. We find that, using a commodity RNIC, we can achieve up to 2.6× and 35× speed-up versus state-of-the-art RDMA approaches, for key-value get operations under uncontended and contended settings, respectively, while allowing applications to gain failure resiliency to OS and process crashes. We believe that this work opens the door for a wide variety of innovations in RNIC offloading which, in turn, can help guide the evolution of the RDMA standard.

RedN is available at https://redn.io.

References

[1] Agilio CX SmartNICs. https://www.netronome.com/products/agilio-cx/.

[2] Catapult. https://www.microsoft.com/en-us/research/project/project-catapult/.

[3] Cavium-Xpliant. https://www.openswitch.net/cavium/.

[4] ConnectX series. https://www.mellanox.com/products/ethernet/connectx-smartnic.

[5] Dynamically Connected (DC) QPs. https://docs.mellanox.com/display/rdmacore50/DynamicallyConnected(DC)QPs.

[6] ibv_modify_qp_rate_limit(3) - Linux man page. https://man7.org/linux/man-pages/man3/ibv_modify_qp_rate_limit.3.html.

[7] Intel Ethernet 800 Series Network Adapters. https://www.intel.com/content/www/us/en/products/docs/network-io/ethernet/network-adapters/ethernet-800-series-network-adapters/e810-cqda1-100gbe-brief.html.

[8] Intel Optane DC Persistent Memory - Product Brief. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-dc-persistent-memory-brief.pdf.

[9] LibVMA. https://github.com/Mellanox/libvma/wiki/Architecture.

[10] LiquidIO II SmartNICs. https://www.marvell.com/products/ethernet-adapters-and-controllers/liquidio-smart-nics/liquidio-ii-smart-nics.html.

[11] Mellanox BlueField. https://www.mellanox.com/products/bluefield-overview.

[12] Mellanox PCX. https://github.com/Mellanox/pcx/tree/master/config.

[13] Mellanox store. http://store.mellanox.com/.

[14] NetFPGA platform. https://netfpga.org/.

[15] RDMA RFC. https://tools.ietf.org/html/rfc5040.

[16] rsocket(7) - Linux man page. https://linux.die.net/man/7/rsocket.

[17] Stingray. https://www.broadcom.com/products/ethernet-connectivity/smartnic.

[18] M. K. Aguilera, N. Ben-David, R. Guerraoui, V. Marathe, and I. Zablotchi. The Impact of RDMA on Agreement. *arXiv preprint arXiv:1905.12143*, 2019.

[19] T. E. Anderson, M. Canini, J. Kim, D. Kostić, Y. Kwon, S. Peter, W. Reda, H. N. Schuh, and E. Witchel. Assise: Performance and Availability via NVM Colocation in a Distributed File System. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.

[20] O. Cardona. Towards Hyperscale High Performance Computing with RDMA, 2019. https://pc.nanog.org/static/published/meetings/NANOG76/1999/20190612_Cardona_Towards_Hyperscale_High_v1.pdf.

[21] S. Dolan. mov is Turing-complete. *Cl. Cam. Ac. Uk*, pages 1–4, 2013.

[22] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, 2014.

[23] H. Eran, L. Zeno, M. Tork, G. Malka, and M. Silberstein. NICA: An Infrastructure for Inline Acceleration of Network Applications. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 345–362, 2019.

[24] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 371–384, 2013.

[25] M. Gabbrielli and S. Martini. *Programming Languages: Principles and Paradigms*, page 145. Undergraduate Topics in Computer Science. Springer London, 2010.

[26] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 295–306. ACM, 2014.

[27] M. Kazhamiaka, B. Memon, C. Kankanamge, S. Sahu, S. Rizvi, B. Wong, and K. Daudjee. Sift: resource-efficient consensus with RDMA. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, pages 260–271, 2019.

[28] D. Kim, A. Memaripour, A. Badam, Y. Zhu, H. H. Liu, J. Padhye, S. Raindel, S. Swanson, V. Sekar, and S. Seshan. Hyperloop: group-based NIC-offloading to accelerate replicated transactions in multi-tenant storage systems. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 297–312, 2018.

[29] B. Li, T. Cui, Z. Wang, W. Bai, and L. Zhang. Socks-Direct: Datacenter sockets can be fast and compatible. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 90–103. 2019.

[30] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 137–152, 2017.

[31] M. Liu, T. Cui, H. Schuh, A. Krishnamurthy, S. Peter, and K. Gupta. Offloading distributed applications onto SmartNICs using iPipe. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 318–333. 2019.

[32] M. Liu, S. Peter, A. Krishnamurthy, and P. M. Phothilimthana. E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 363–378, 2019.

[33] Y. Lu, J. Shu, Y. Chen, and T. Li. Octopus: An RDMA-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 773–785, 2017.

[34] Mellanox RDMA Aware Networks Programming User Manual. https://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf.

[35] C. Mitchell, Y. Geng, and J. Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 103–114, 2013.

[36] P. M. Phothilimthana, M. Liu, A. Kaufmann, S. Peter, R. Bodik, and T. Anderson. Floem: A Programming System for NIC-Accelerated Network Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 663–679, 2018.

[37] M. Poke and T. Hoefler. Dare: High-performance State Machine Replication on RDMA Networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 107–118. ACM, 2015.

[38] A. Rosenbaum. Multiprocess Sharing of RDMA Resources, 2018. https://openfabrics.org/images/2018workshop/presentations/103_ARosenbaum_Multi-ProcessSharing.pdf.

[39] D. Sidler, Z. Wang, M. Chiosa, A. Kulkarni, and G. Alonso. StRoM: Smart Remote Memory. *Proceedings of the Fifteenth EuroSys Conference*, 2020.

[40] A. K. Simpson, A. Szekeres, J. Nelson, and I. Zhang. Securing RDMA for High-Performance Datacenter Storage Systems. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, 2020.

[41] C. Wang, J. Jiang, X. Chen, N. Yi, and H. Cui. APUS: Fast and Scalable Paxos on RDMA. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 94–107, 2017.

[42] X. Wei, Z. Dong, R. Chen, and H. Chen. Deconstructing RDMA-enabled distributed transactions: Hybrid is better! In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 233–251, 2018.

[43] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 87–104, 2015.

[44] J. Yang, J. Izraelevitz, and S. Swanson. Orion: A distributed file system for non-volatile main memory and RDMA-capable networks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 221–234, 2019.

[45] D. Y. Yoon, M. Chowdhury, and B. Mozafari. Distributed lock management with RDMA: decentralization without starvation. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1571–1586, 2018.

[46] T. Ziegler, S. Tumkur Vani, C. Binnig, R. Fonseca, and T. Kraska. Designing distributed tree-based index structures for fast RDMA-capable networks. In *Proceedings of the 2019 International Conference on Management of Data*, pages 741–758, 2019.

Appendix A   Turing completeness sketch

To show that RDMA is turing complete, we need to establish that RDMA has the following three properties:

1. Can read/write arbitrary amounts of memory.
2. Has conditional branching (e.g., if & else statements).
3. Allows nontermination.

Our paper already demonstrates that these properties can be satisfied using our constructs but, for completeness, we also analogize our system with x86 assembly instructions that have been proven to be capable of simulating a Turing machine. Dolan [21] demonstrated that this is in fact possible using just the x86 mov instruction. As such, we need to prove that RDMA has sufficient expressive power to emulate the mov instruction.

A.1   Emulating the **x86 mov** instruction

To provide an RDMA implementation for mov, we first need to consider the different addressing modes used by Dolan [21] to simulate a Turing machine. The addressing mode describes how a memory location is specified in the mov operands.

Table 7 shows a list of all required addressing modes, their x86 syntax, and one possible implementation for each with RDMA. *R* operands denote registers but, since RDMA operations can only perform memory-to-memory transfers, we assume these registers are stored in memory. For simplicity, we only focus on mov instructions used to perform *loads* but note that *stores* can be implemented in a similar manner.

For *immediate* addressing, the operand is part of the instruction and is passed directly to register R_{dst}. This can be implemented simply using an WRITEIMM which takes a constant in its *immediate* parameter and writes it to a specified memory location (register R_{dst} in this case). To perform more complex operations, *indirect* allows mov to use the value of

the operand as a memory address. This enables the dynamic modification of the address at runtime, since it depends on the contents of the register when the instruction is executed. To implement this, we use two write operations with doorbell ordering (refer to §3.1 for a discussion of our ordering modes). The first WRITE changes the *source address* attribute of the second WRITE operation to the value in register R_{src}. This allows the second WRITE operation to write to register R_{dst} using the value at the memory address pointed to by R_{src}. Lastly, *indexed* addressing allows us to add an offset (R_{off}) to the address in register R_{src}. This can be done by simply performing an RDMA ADD operation between the two writes with doorbell ordering, in order to add the offset register value R_{off} to R_{src}. This allows us to finally write the value $[R_{src} + R_{off}]$ to R_{dst}. With these three implementations, we showcase that RDMA can in fact emulate all the required mov instruction variants.

A.2   Allowing nontermination

To simulate a real Turing machine, we need to also satisfy the code nontermination requirement. In the x86 architecture, this can be achieved via an unconditional jump [21] that loops back to the start of the program. For RDMA, this can also be achieved by having the CPU re-post the WRs after they are executed. While this is sufficient for Turing completeness it, nevertheless, wastes additional CPU cycles and can also impact latency if CPU cores are busy or unable to keep up with WR execution. As an alternative, RedN provides a way to loop back without any CPU interaction by relying on WAIT and ENABLE to recycle RDMA WRs (as described in §3.4). Regardless of which approach is employed, RDMA is capable of performing an unconditional jump to the beginning of the program. This means that we can emulate all x86 instructions used by Dolan [21] for simulating a Turing machine.
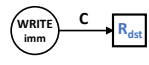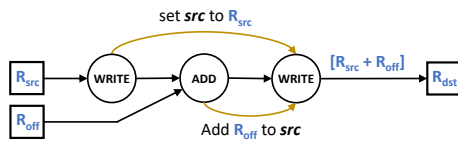
Addressing mode	x86 syntax	RedN equivalent
Immediate	mov R_{dst}, C	
Indirect	mov R_{dst}, $[R_{src}]$	
Indexed	mov R_{dst}, $[R_{src} + R_{off}]$	

Table 7: Addressing modes for the x86 mov instruction and their RDMA implementation in RedN.

FlexTOE: Flexible TCP Offload with Fine-Grained Parallelism

Rajath Shashidhara[1]   Tim Stamler[2]   Antoine Kaufmann[3]   Simon Peter[1]

[1]University of Washington   [2]UT Austin   [3]MPI-SWS

Abstract

FlexTOE is a flexible, yet high-performance TCP offload engine (TOE) to SmartNICs. FlexTOE eliminates almost all host data-path TCP processing and is fully customizable. FlexTOE interoperates well with other TCP stacks, is robust under adverse network conditions, and supports POSIX sockets.

FlexTOE focuses on data-path offload of established connections, avoiding complex control logic and packet buffering in the NIC. FlexTOE leverages fine-grained parallelization of the TCP data-path and segment reordering for high performance on wimpy SmartNIC architectures, while remaining flexible via a modular design. We compare FlexTOE on an Agilio-CX40 to host TCP stacks Linux and TAS, and to the Chelsio Terminator TOE. We find that Memcached scales up to 38% better on FlexTOE versus TAS, while saving up to 81% host CPU cycles versus Chelsio. FlexTOE provides competitive performance for RPCs, even with wimpy SmartNICs. FlexTOE cuts 99.99th-percentile RPC RTT by 3.2× and 50% versus Chelsio and TAS, respectively. FlexTOE's data-path parallelism generalizes across hardware architectures, improving single connection RPC throughput up to 2.4× on x86 and 4× on BlueField. FlexTOE supports C and XDP programs written in eBPF. It allows us to implement popular data center transport features, such as TCP tracing, packet filtering and capture, VLAN stripping, flow classification, firewalling, and connection splicing.

1   Introduction

TCP remains the default protocol in many networks, even as its CPU overhead is increasingly a burden to application performance [3, 17, 46]. A long line of improvements to software TCP stack architecture has reduced overheads: Careful packet steering improves cache-locality for multi-cores [17, 24, 45], kernel-bypass enables safe direct NIC access from user-space [3, 46], application libraries avoid system calls for common socket operations [17], and fast-paths drastically reduce TCP processing overheads [19]. Yet, even with these optimizations, communication-intensive applications spend up to 48% of per-CPU cycles in the TCP stack and NIC driver (§2.1).

Offload promises further reduction of CPU overhead. While moving parts of TCP processing, such as checksum and segmentation, into the NIC is commonplace [54], full TCP offload engines (TOEs) [6, 7, 33] have so far failed to find widespread adoption. A primary reason is that fixed offloads [56] limit protocol evolution after deployment [9, 29, 36]. Tonic [2] provides building blocks for flexible transport protocol offload to FPGA-SmartNICs, but FPGA development is still difficult and slow.

We present FlexTOE, a high-performance, yet flexible offload of the widely-used TCP protocol. FlexTOE focuses on

scenarios that are common in data centers, where connections are long-lived and small transfers are common [29]. FlexTOE offloads the TCP data-path to a network processor (NPU) based SmartNIC, enabling full customization of transport logic and flexibility to implement data-path features whose requirements change frequently in data centers. Applications interface directly but transparently with the FlexTOE datapath through the *libTOE* library that implements POSIX sockets, while FlexTOE offloads all TCP data-path processing (§2.1).

TCP data-path offload to SmartNICs is challenging. Smart-NICs support only restrictive programming models with stringent per-packet time budgets and are geared towards massive parallelism with wimpy cores [26]. They often lack timers, as well as floating-point and other computational support, such as division. Finally, offload has to mask high-latency operations that cross PCIe. On the other hand, TCP requires computationally intensive and stateful code paths to track in-flight segments, for reassembly and retransmission, and to perform congestion control [2]. For each connection, the TCP data-path needs to provide low processing tail latency and high throughput and is also extremely sensitive to reordering.

Resolving the gap between TCP's requirements and Smart-NIC hardware capabilities requires careful offload design to efficiently utilize SmartNIC capabilities. Targeting FlexTOE at the TCP data-path of established connections avoids complex control logic in the NIC. FlexTOE's offloaded data-path is one-shot for each TCP segment—segments are never buffered in the NIC. Instead, per-socket buffers are kept in per-process host memory where libTOE interacts with them directly. Connection management, retransmission, and congestion control are part of a separate control-plane, which executes in its own protection domain, either on control cores of the SmartNIC or on the host. To provide scalability and flexibility, we decompose the TCP data-path into fine-grained modules that keep private state and communicate explicitly. Like microservices [29], FlexTOE modules leverage a data-parallel execution model that maximizes SmartNIC resource use **and** simplifies customization. We organize FlexTOE modules into a *data-parallel computation pipeline*. We also *reorder* segments on-the-fly to support parallel, out-of-order processing of pipeline stages, while enforcing in-order TCP segment delivery. To our knowledge, no prior work attempting full TCP data-path offload to NPU SmartNICs exists.

We make the following contributions:

- We characterize the CPU overhead of TCP data-path processing for common data center applications (§2.1). Our analysis shows that up to 48% of per-CPU cycles are spent in TCP data-path processing, even with optimized TCP stacks.

- We present FlexTOE, a flexible, high-performance TCP of-fload engine (§3). FlexTOE leverages data-path processing with fine-grained parallelism for performance, but remains flexible via a modular design. We show how to decompose TCP into a data-path and a control-plane, and the data-path into a data-parallel pipeline of processing modules to hide SmartNIC processing and data access latencies.
- We implement FlexTOE on the Netronome Agilio-CX40 NPU SmartNIC architecture, as well as x86 and Mellanox BlueField (§4). Using FlexTOE design principles, we are the first to demonstrate that NPU SmartNICs can support scal-able, yet flexible TCP data-path offload. Our code is avail-able at https://tcp-acceleration-service.github.io/FlexTOE.
- We evaluate FlexTOE on a range of workloads and com-pare to Linux, the high-performance TAS [19] network stack, and a Chelsio Terminator TOE [6] (§5). We find that the Memcached [32] key-value store scales throughput up to 38% better on FlexTOE than using TAS, while saving up to 81% host CPU cycles versus Chelsio. FlexTOE cuts 99.99th-percentile RPC RTT by 3.2× and 50% versus Chelsio and TAS respectively, 27% higher throughput than Chel-sio for bidirectional long flows, and an order of magnitude higher throughput under 2% packet loss than Chelsio. We extend the FlexTOE data-path with debugging and auditing functionality to demonstrate flexibility. FlexTOE maintains high performance when interoperating with other network stacks. FlexTOE's data-path parallelism generalizes across platforms, improving single connection RPC throughput up to 2.4× on x86 and 4× on BlueField.

2 Background

We motivate FlexTOE by analyzing TCP host CPU processing overheads of related approaches (§2.1). We then place Flex-TOE in context of this and further related work (§2.2). Finally, we survey the relevant *on-path* SmartNIC architecture (§2.3).

2.1 TCP Impact on Host CPU Performance

We quantify the impact of different TCP processing approaches on host CPU performance in terms of CPU overhead, execu-tion efficiency, and cache footprint, when processing common RPC-based workloads. We do so by instrumenting a single-threaded Memcached [32] server application using hardware performance counters (cf. §5 for details of our testbed). We use the popular memtier_benchmark [51] to generate the client load, consisting of 32 B keys and values, using as many clients as necessary to saturate the server, executing closed-loop KV transactions on persistent connections. Table 1 shows a breakdown of our server-side results, for each Memcached request-response pair, into NIC driver, TCP/IP stack, POSIX sockets, Memcached application, and other factors.

In-kernel. Linux's TCP stack is versatile but bulky, lead-ing to a large cache footprint, inefficient execution, and high CPU overhead. Stateless offloads [54], such as segmentation

Module	Linux		Chelsio		TAS		FlexTOE	
	kc	%	kc	%	kc	%	kc	%
NIC driver	0.71	6	1.28	14	0.18	5	0	0
TCP/IP stack	4.25	35	0.40	4	1.44	43	0	0
POSIX sockets	2.48	21	2.61	29	0.79	23	0.74	44
Application	1.26	10	1.31	16	0.85	26	0.89	53
Other	3.42	28	3.28	37	0.09	3	0.04	3
Total	12.13	100	8.89	100	3.34	100	1.67	100
Retiring	4.60	38	2.43	27	1.66	48	0.77	46
Frontend bound	3.53	29	1.52	17	0.46	13	0.34	21
Backend bound	3.40	28	4.68	53	1.24	36	0.46	27
Bad speculation	0.55	5	0.26	3	0.13	4	0.09	6
Instructions (k)	16.18		8.14		6.26		2.93	
IPC	1.33		0.92		1.85		1.75	
Icache (KB)	47.50		73.43		39.75		19.00	

Table 1. Per-request CPU impact of TCP processing.

and generic receive offload [12], reduce overhead for large transfers, but they have minimal impact on RPC workloads dominated by short flows. We find that Linux executes 12.13 kc per Memcached request on average, with only 10% spent in the application. Not only does Linux have a high instruction and instruction cache (Icache) footprint, but privilege mode switches, scattered global state, and coarse-grained locking lead to 62% of all cycles spent in instruction fetch stalls (fron-tend bound), cache and TLB misses (backend bound), and branch mispredictions (cf. [19]). These inefficiencies result in 1.33 instructions per cycle (IPC), leveraging only 33% of our 4-way issue CPU architecture. Linux is, in principle, easy to modify, but kernel code development is complex and security sensitive. Hence, introducing optimizations and new network functionality to the kernel is often slow [29, 42, 43].

Kernel-bypass. Kernel-bypass, such as in mTCP [17] and Arrakis [46], eliminates kernel overheads by entrusting the TCP stack to the application, but it has security implica-tions [52]. TAS [19] and Snap [29] instead execute a protected user-mode TCP stack on dedicated cores, retaining security and performance. By eliminating kernel calls, TAS spends only 800 cycles in the socket API—31% of Linux's API over-head. TAS also reduces TCP stack overhead to 34% of Linux. TAS reduces Icache footprint, front and back-end CPU stalls, improving IPC by 40% versus Linux, and reducing the total per-request CPU impact to 27% of Linux. However, kernel-bypass still has significant overhead. Only 26% of per-request cycles are spent in Memcached—the remainder is spent in TAS (breakdown in §C).

Inflexible TCP offload. TCP offload can eliminate host CPU overhead for TCP processing. Indeed, TOEs [7] that of-fload the TCP data-path to the NIC have existed for a long time. Existing approaches, such as the Chelsio Terminator [6], hard-wire the TCP offload. The resulting inflexibility prevents data center operators from adapting the TOE to their needs

and leads to a slow upgrade path due to long hardware development cycles. For example, the Chelsio Terminator line has been slow to adapt to RPC-based data center workloads.

Chelsio's inflexbility shows in our analysis. Despite drastically reducing the host TCP processing cycles to 10% of Linux and 28% of TAS, Chelsio's TOE only modestly reduces the total per-request CPU cycles of Memcached by 27% versus Linux and inflates them by 2.6× versus TAS. Chelsio's design requires interaction through the Linux kernel, leading to a similar execution profile despite executing 50% fewer host instructions per request. In addition, Chelsio requires a sophisticated TOE NIC driver, with complex buffer management and synchronization. Chelsio's design is inefficient for RPC processing and leaves only 16% of the total per-request cycles to Memcached—6% more than Linux and 10% fewer than TAS.

FlexTOE. FlexTOE eliminates all host TCP stack overheads. FlexTOE's instruction (and Icache) footprint is at least 2× lower than the other stacks, leading to an execution profile similar to TAS, where 46% of all cycles are spent retiring instructions. In addition, 53% of all cycles can be spent in Memcached—an improvement of 2× versus TAS, the next best solution. The remaining cycles are spent in the POSIX sockets API, which cannot be eliminated with TCP offload.

FlexTOE is also flexible, allowing operators to modify the TOE at will. For example, we have modified the TCP data-path many times, implementing many features that require TOE modification, including scalable socket API implementations [24, 45], congestion control protocols [1, 34], scalable flow scheduling [53], scalable PCIe communication protocols [44], TCP tracing [13], packet filtering and capture (tcp-dump and PCAP), VLAN stripping, programmable flow classification (eBPF [30]), firewalling, and connection splicing similar to AccelTCP [37]. All of these features are desirable in data centers and are adapted frequently.

2.2 Related Work

Beyond the TCP implementations covered in §2.1, we cover here further related work in SmartNIC offload, parallel packet processing, and API and network protocol specialization.

SmartNIC offload. On-path SmartNICs (§2.3), based on network processor units (NPUs) and FPGAs, provide a suitable substrate for flexible offload. Arsenic [47] is an early example of flexible packet multiplexing on a SmartNIC. Microsoft's Catapult [48] offloads network management, while Dagger [22] offloads RPC processing to FPGA-SmartNICs. Neither offloads a transport protocol, like TCP. AccelTCP [37] offloads TCP connection management and splicing [28] to NPU-SmartNICs, but keeps the TCP data-path on the host using mTCP [17]. Tonic [2] demonstrates in simulation that high-performance, flexible TCP transmission offload might be possible, but it stops short of implementing full TCP data-path offload (including receiver processing) in a non-simulated environment. LineFS [20] offloads a distributed file system to an off-path

SmartNIC, leveraging parallelization to hide execution latencies of wimpy SmartNIC CPUs and data access across PCIe. Taking inspiration from Tonic and LineFS, but also from actor, and microservice-based approaches presented in iPipe [26], E3 [27], and Click [23, 38], FlexTOE shows how to decompose the TCP data-path into a fine-grained data-parallel pipeline to support full and flexible offload to on-path NPU-SmartNICs.

Parallel packet processing. RouteBricks [8] parallelizes across cores and cluster nodes for high-performance routing, achieving high line-rates but remaining flexible via software programmability. Routing relies on read-mostly state and is simple compared to TCP. FlexTOE applies fine-grained parallelization to complex, stateful code paths.

Specialized APIs and protocols. Another approach to lower CPU utilization is specialization. R2P2 [21] is a UDP-based protocol for remote procedure calls (RPCs) optimized for efficient and parallel processing, both at the end-hosts and in the network. eRPC [18] goes a step further and co-designs an RPC protocol and API with a kernel-bypass network stack to minimize CPU overhead per RPC. RDMA [49] is a popular combination of a networking API, protocol, and a (typically hardware) network stack. iWARP [50], in particular, leverages a TCP stack underneath RDMA, offloading both. These approaches improve processing efficiency, but at the cost of requiring application re-design, all-or-nothing deployments, and operational issues at scale [11], often due to inflexibility [36, 56]. FlexTOE instead offloads the TCP protocol in a flexible manner by relying on SmartNICs. Upper-layer protocols, such as iWARP, can also be implemented using FlexTOE.

2.3 On-path SmartNIC Architecture

On-path SmartNICs[1], such as Marvell Octeon [5], Pensando Capri [10, 55], and Netronome Agilio [39, 40], support massively parallel packet processing with a large pool of flow processing cores (FPCs), but they lack efficient support for sophisticated program control flow and complex computation [26].

We explore offload to the NFP-4000 NPU, used in Netronome Agilio CX SmartNICs [39]. We show the relevant architecture in Figure 1. Like other on-path SmartNICs, FPCs are organized into islands with local memory and processing resources, akin to NUMA domains. Islands are connected in a mesh via a high-bandwidth interconnect (arrows in Figure 1).
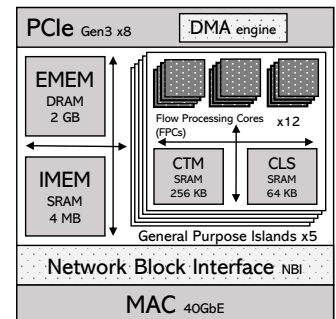
Figure 1. NFP-4000 overview.

---

[1]Mellanox BlueField [31] and Broadcom Stingray [4] are off-path SmartNICs that are not optimized for packet processing [26].

The *PCIe* island has up to two PCIe Gen3 x8 interfaces and a DMA engine exposing DMA transaction queues [41]. FPCs can issue up to 256 asynchronous DMA transactions to perform IO between host and NIC memory. The *MAC* island supports up to two 40 Gbps Ethernet interfaces, accessed via a *network block interface* (NBI).

Flow Processing Cores (FPCs). 60 FPCs are grouped into five general-purpose islands (each containing 12 FPCs). Each FPC is an independent 32-bit core at 800 MHz with 8 hardware threads, 32 KB instruction memory, 4 KB data memory, and CRC acceleration. While FPCs have strong data flow processing capabilities, they have small codestores, lack timers, as well as floating-point and other complex computational support, such as division. This makes them unsuitable to execute computationally and control intensive TCP functionality, such as congestion, connection, and complex retransmission control. For example, congestion avoidance involves computing an ECN-ratio (gradient). We found that it takes 1,500 cycles (1.9 μs) per RTT to perform this computation on FPCs.

Memory. The NFP-4000 includes multiple memories of various sizes and performance characteristics. General-purpose islands have 64KB of island-local scratch (*CLS*) and 256 KB of island target memory (*CTM*), with access latencies of up to 100 cycles from island-local FPCs for data processing and transfer, respectively. The internal memory unit (*IMEM*) provides 4 MB of SRAM with an access latency of up to 250 cycles. The external memory unit (*EMEM*) provides 2 GB of DRAM, fronted by a 3 MB SRAM cache, with up to 500 cycles latency.

Implications for flexible offload. The NFP-4000 supports a broad range of protocols, but the computation and memory restrictions require careful offload design. As FPCs are wimpy and memory latencies high, sequential instruction execution is much slower than on host processors. Conventional run-to-completion processing that assigns entire connections to cores [3, 17, 19] results in poor per-connection throughput and latency. In some cases, it is beyond the feasible instruction and memory footprint. Instead, an efficient offload needs to leverage more fine-grained parallelism to limit the per-core compute and memory footprint.

3 FlexTOE Design

In addition to flexibility, FlexTOE has the following goals:

- **Low tail latency and high throughput**. Modern datacenter network loads consist of short and long flows. Short flows, driven by remote procedure calls, require low tail completion time, while long flows benefit from high throughput. FlexTOE shall provide both.
- **Scalability**. The number of network flows and application contexts that servers must handle simultaneously is increasing. FlexTOE shall scale with this demand.

To achieve these goals and overcome SmartNIC hardware limitations, we propose three design principles:

1. **One-shot data-path offload.** We focus offload on the TCP RX/TX data-path, eliminating complex control, compute, and state, thereby also enabling fine-grained parallelization. Further, our data-path offload is one-shot for each TCP segment. Segments are never buffered on the NIC, vastly simplifying SmartNIC memory management.
2. **Modularity.** We decompose the TCP data-path into fine-grained, customizable modules that keep private state and communicate explicitly. New TCP extensions can be implemented as modules and hooked into the data-flow, simplifying development and integration.
3. **Fine-grained parallelism.** We organize the data-path modules into a data-parallel computation pipeline that maximizes SmartNIC resource use. We map stages to FPCs, allowing us to fully utilize all FPC resources. We employ TCP segment sequencing and reordering to support parallel, out-of-order processing of pipeline stages, while enforcing in-order segment delivery.

Decomposing TCP for offload. We use the TAS host TCP stack architecture [19] as a starting point. TAS splits TCP processing into three components: a data-path, a control-plane, and an application library. The data-path is responsible for scalable data transport of established connections: TCP segmentation, loss detection and recovery, rate control, payload transfer between socket buffers and the network, and application notifications. The control-plane handles connection and context management, congestion control, and complex recovery involving timeouts. Finally, the application library intercepts POSIX socket API calls and interacts with control-plane and data-path using dedicated context queues in shared memory. Data-path and control-plane execute in their own protection domains on dedicated cores, isolated from untrusted applications, and communicate through efficient message passing queues.

FlexTOE offload architecture. In FlexTOE we adapt this architecture for offload, by designing and integrating a *data-path running efficiently* on the SmartNIC (§3.1). The FlexTOE control-plane can run on the host or on a SmartNIC control CPU, with the same functionality as in TAS (cf. §D). The FlexTOE control-plane additionally manages the SmartNIC data-path resources. Similarly, our application library (libTOE) intercepts POSIX socket calls and is dynamically linked to unmodified processes that use FlexTOE, and communicates directly with the data-path.

Figure 2 shows the offload architecture of FlexTOE, with a host control-plane (each box is a protection domain). libTOE, data-path, and control-plane communicate via pairs of *context queues* (CTX-Qs), one for each communication direction. CTX-Qs leverage PCIe DMA and MMIO or shared memory for SmartNIC-host and intra-host communication, respectively.
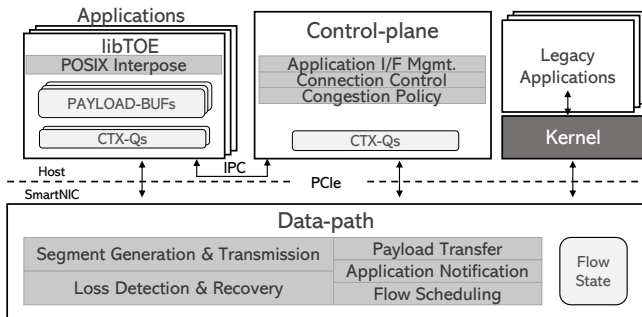
Figure 2. FlexTOE offload architecture (host control-plane).

FlexTOE supports per-thread context queues for scalability. Each TCP socket keeps receive and transmit payload buffers (PAYLOAD-BUFs) in host memory. libTOE appends data for transmission into the per-socket TX PAYLOAD-BUF and notifies the data-path using a thread-local CTX-Q. The data-path appends received segments to the socket's RX PAYLOAD-BUF after reassembly and libTOE is notified via the same thread-local CTX-Q. Non-FlexTOE traffic is forwarded to the Linux kernel, which legacy applications may use simultaneously.

3.1 TCP Data-path Parallelization

To provide high offload performance using relatively wimpy SmartNIC FPCs, FlexTOE has to leverage all available parallelism within the TCP data-path. In this section, we analyze the TAS host TCP data-path to investigate what parallelism can be extracted. In particular, the TCP data-path in TAS has the following three workflows:

- **Host control (HC)**: When an application wants to transmit data, executes control operations on a socket, or when retransmission is necessary, the data-path must update the connection's transmit and receive windows accordingly.
- **Transmit (TX)**: When a TCP connection is ready to send—based on congestion and flow control—the data-path prepares a segment for transmission, fetching its payload from a socket transmit buffer and sending it out to the MAC.
- **Receive (RX)**: For each received segment of an established connection, the data-path must perform byte-stream reassembly—advance the TCP window, determine the segment's position in the socket receive buffer, generate an acknowledgment to the sender, and, finally, notify the application. If the received segment acknowledges previously transmitted segments, the data-path must also free the relevant payload in the socket transmit buffer.

Host TCP stacks, such as Linux or TAS, typically process each workflow to completion in a critical section accessing a shared per-connection state structure. HC workflows are typically processed on the program threads that trigger them, while TX and RX are typically triggered by NIC interrupts and processed on high-priority (kernel or dedicated) threads.

For efficient offload, we decompose this data-path into an up to five-stage parallel pipeline of processing modules: *pre-processing*, *protocol*, *post-processing*, *DMA*, and *context queue*
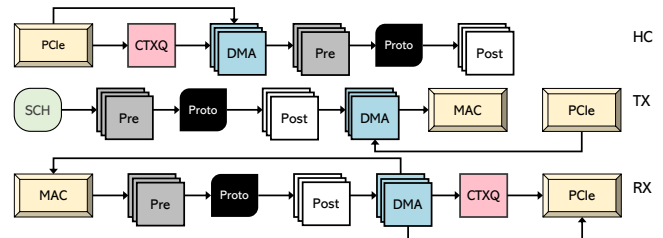
Figure 3. Per-connection data-path workflows. *Protocol* is atomic. Other stages may be replicated for parallelism.
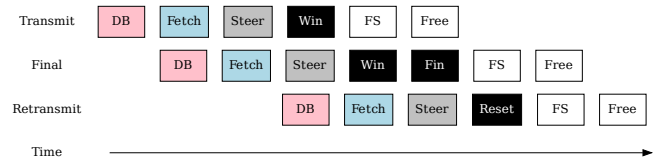
Figure 4. HC pipeline: Transmit, FIN, and retransmit.

(Figure 3). Accordingly, we partition connection state into module-local state (cf. §A). The pipeline stages are chosen to maximize data-path parallelism. Pre-processing accesses connection identifiers such as MAC and IP addresses for segment header preparation and filtering. The post-processing block handles application interface parameters, such as socket buffer addresses and context queues. These parameters are read-only after connection establishment and enable coordination-free scaling. Congestion control statistics are collected by the post-processor, but are only read by forward stages and can be updated out-of-order (updates commute). The protocol stage executes data-path code that must atomically modify protocol state, such as sequence numbers and socket buffer positions. It is the only *pipeline hazard*—it cannot execute in parallel with other stages. The DMA stage is stateless, while context queue stages may be sharded. Both conduct high-latency PCIe transactions and are thus separate stages that execute in parallel and scale independently.

We run pipeline stages on dedicated FPCs that utilize local memory for their portion of the connection state. Pipelining allows us to execute the data-path in parallel. It also allows us to replicate processing-intensive pipeline stages to scale to additional FPCs. With the exception of protocol processing, which is atomic per connection, all pipeline stages are replicated. To concurrently process multiple connections, we also replicate the entire pipeline. To keep flow state local, each pipeline handles a fixed *flow-group*, determined by a hash on the flow's 4-tuple (the flow's protocol type is ignored—it must be TCP). We now describe how we parallelize each data-path workflow by decomposing it into these pipeline stages.

3.1.1 Host Control (HC). HC processing is triggered by a PCIe doorbell (DB) sent via memory-mapped IO (MMIO) by the host to the context queue stage. Figure 4 shows the HC pipeline for two transmits (the second transmit closes the connection) triggered by libTOE, and a retransmit triggered by the control-plane. HC requests may be batched.
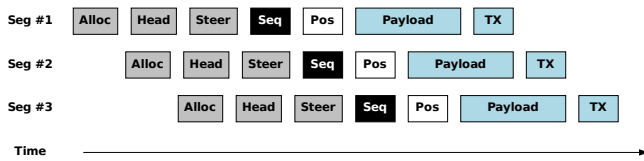
Figure 5. TX pipeline sending 3 segments.

Figure 6. RX pipeline receiving 3 segments, 1 out of order.

The context queue stage polls for DBs. In response to a DB, the stage allocates a descriptor buffer from a pool in NIC memory. The limited pool size flow-controls host interactions. If allocation fails, processing stops and is retried later. Otherwise, the DMA stage fetches the descriptor from the host context queue into the buffer (Fetch). The pre-processor reads the descriptor, determines the flow-group, and routes to the appropriate protocol stage (Steer). The protocol stage updates connection receive and transmit windows (Win). If the HC descriptor contains a connection-close indication, the protocol stage also marks the connection as FIN (Fin). When the transmit window expands due to the application sending data for transmission, the post-processor updates the flow scheduler (FS) and returns the descriptor to the pool (Free).

Retransmissions in response to timeouts are triggered by the control-plane and processed the same as other HC events (fast retransmits due to duplicate ACKs are described in §3.1.3). The protocol stage resets the transmission state (Reset) to the last ACKed sequence number (go-back-N retransmission).

3.1.2 Transmit (TX). Transmission is triggered by the flow scheduler (SCH) when a connection can send segments. Figure 5 shows the TX pipeline for 3 example segments.

The pre-processor allocates a segment in NIC memory (Alloc), prepares Ethernet and IP headers (Head), and steers the segment to the flow-group's protocol stage (Steer). The protocol stage assigns a TCP sequence number based on connection state and determines the transmit offset in the host socket transmit buffer (Seq). The post-processor determines the socket transmit buffer address in host memory (Pos). The DMA stage fetches the host payload into the segment (Payload). After DMA completes, it issues the segment to the NBI (TX), which transmits and frees it.

3.1.3 Receive (RX). Figure 6 shows the RX pipeline for 3 example segments, where segment #3 arrives out of order.

Pre-processing. The pre-processor first validates the segment header (Val). Non-data-path segments[2] are filtered and forwarded to the control-plane. Otherwise, the pre-processor determines the connection index based on the segment's 4-tuple (Id) that is used by later stages to access connection state. The pre-processor generates a *header summary* (Sum), including only relevant header fields required by later pipeline stages and steers the summary and connection identifier to the protocol stage of its flow-group (Steer).

---

[2]*Data-path segments* have any of the ACK, FIN, PSH, ECE, and CWR flags and they may have the timestamp option.
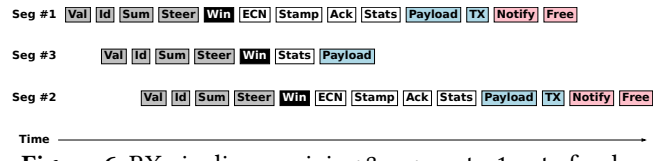
Protocol. Based on the header summary, the protocol stage updates the connection's sequence and acknowledgment numbers, the transmit window, and determines the segment's position in the host socket receive payload buffer, trimming the payload to fit the receive window if necessary (Win). The protocol stage also tracks duplicate ACKs and triggers fast retransmissions if necessary, by resetting the transmission state to the last acknowledged position. Finally, it forwards a snapshot of relevant connection state to post-processing.

Out-of-order arrivals (segment #3 in Figure 6) need special treatment. Like TAS [19], we track one out-of-order interval in the receive window, allowing the protocol stage to perform reassembly directly within the host socket receive buffer. We merge out-of-order segments within the interval in the host receive buffer. Segments outside of the interval are dropped and generate acknowledgments with the expected sequence number to trigger retransmissions at the sender. This design performs well under loss (cf. §5.3).

Post-processing. The post-processor prepares an acknowledgment segment (Ack). FlexTOE provides explicit congestion notification (ECN) feedback and accurate timestamps for RTT estimation (Stamp) in acknowledgments. It also collects congestion control and transmit window statistics, which it sends to the control-plane and flow scheduler (Stats). Finally, it determines the physical address of the host socket receive buffer, payload offset, and length for the DMA stage. If libTOE is to be notified, the post-processor allocates a context queue descriptor with the appropriate notification.

DMA. The DMA stage first enqueues payload DMA descriptors to the PCIe block (Payload). After payload DMA completes, the DMA stage forwards the notification descriptor to the context queue stage. Simultaneously, it sends the prepared acknowledgment segment to the NBI (TX), which frees it after transmission. This ordering is necessary to prevent the host and the peer from receiving notifications before the data transfer to the host socket receive buffer is complete.

Context queue. If necessary, the context queue stage allocates an entry on the context queue and issues the context queue descriptor DMA to notify libTOE of new payload (Notify) and frees the internal descriptor buffer (Free).

3.2 Sequencing and Reordering

TCP requires that segments of the same connection are processed in-order for receiver loss detection. However, stages in FlexTOE's data-parallel processing pipeline can have varying processing time and hence may reorder segments. Figure 7
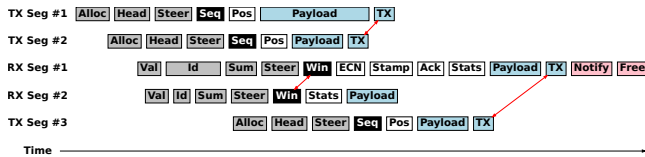
Figure 7. Undesirable pipeline reordering (red arrows).

shows three examples on a bidirectional connection where undesirable segment reordering occurs.

1. **TX.** TX segment #1 stalls in DMA across a congested PCIe link, causing it to be transmitted on the network after TX segment #2, potentially triggering receiver loss detection.

2. **RX.** RX segment #1 stalls in flow identification during pre-processing, entering the protocol stage later than RX segment #2. The protocol stage detects a hole and triggers unnecessary out-of-order processing.

3. **ACK.** TX segment #3 is processed after RX segment #1 in the protocol stage. RX segment #1 generates an ACK, but RX post-processing is complex, resulting in TX segment #3 with a higher sequence number being sent before ACK segment #1.

To avoid reordering, FlexTOE's data-path pipeline sequences and reorders segments if necessary. In particular, we assign a sequence number to each segment entering the pipeline. The parallel pipeline stages can operate on each segment in any order. The protocol stage requires in-order processing and we buffer and re-order segments that arrive out-of-order before admitting them to the protocol stage. Similarly, we buffer and re-order segments for transmission before admitting them to the NBI. We leverage additional FPCs for sequencing, buffering, and reordering.

3.3 Flexibility

Data center networks evolve quickly, requiring TCP stacks to be easily modifiable by operators, not just vendors [29, 42, 43]. Many desirable data center features require TOE modification and are adapted frequently by operators. FlexTOE provides flexibility necessary to implement and maintain these features even beyond host stacks such as TAS, by relying on a programmable SmartNIC. To simplify development and modification of the TCP data-path, FlexTOE provides an extensible, data-parallel pipeline of self-contained modules, similar to the Click [38] extensible router.

Module API. The FlexTOE module API provides developers one-shot access to TCP segments and associated meta-data. Meta-data may be created and forwarded along the pipeline by any module. Modules may also keep private state. For scalability, private state cannot be accessed by other modules or replicas of the same module. Instead, state that may be accessed by further pipeline stages is forwarded as meta-data.

The replication factor of pipeline stages and assignment to FPCs is manual and static in FlexTOE. As long as enough FPCs are available, this approach is acceptable. Operators can determine an appropriate replication factor that yields acceptable TCP processing bandwidth for a pipeline stage via throughput microbenchmarks at deployment. Stages that modify connection state atomically may be deployed by inserting an appropriate steering stage that steers segments of a connection to the module in the atomic stage, holding their state (cf. protocol processing stage in §3.1).

XDP modules. FlexTOE also supports eXpress Data Path (XDP) modules [14–16], implemented in eBPF. XDP modules operate on raw packets, modify them if necessary, and output one of the following result codes: (i) `XDP_PASS`: Forward the packet to the next FlexTOE pipeline stage. (ii) `XDP_DROP`: Drop the packet. (iii) `XDP_TX`: Send the packet out the MAC. (iv) `XDP_REDIRECT`: Redirect the packet to the control-plane.

XDP modules may use BPF maps (arrays, hash tables) to store and modify state atomically [25], which may be modified by the control-plane. For example, a firewall module may store blacklisted IPs in a hash map and the control-plane may add or remove entries dynamically. The module can consult the hash map to determine if a packet is blacklisted and drop it. XDP stages scale like other pipeline stages, by replicating the module. FlexTOE automatically reorders processed segments after a parallel XDP stage (§3.2).

Using these APIs, we modified the FlexTOE data-path many times, implementing the features listed in §2.1 (evaluation in §5.1). Further, ECN feedback and segment timestamping (cf. §3.1.3) are optional TCP features that support our congestion control policies. Operators can remove the associated post-processing modules if they are not needed.

By handling atomicity, parallelization, and ordering concerns, FlexTOE allows complex offloads to be expressed using few lines of code. For example, we implement AccelTCP's connection splicing in 24 lines of eBPF code (cf. Listing 1 in the appendix). The module performs a lookup on the segment 4-tuple in a BPF hashmap. If a match is not found, we forward the segment to the next pipeline stage. Otherwise, we modify the destination MAC and IP addresses, TCP ports, and translate sequence and acknowledgment numbers using offsets configured by the control-plane, based on the connection's initial sequence number. Finally, we transmit. FlexTOE handles sequencing and updating the checksum of the segment. Additionally, when we receive segments with control flags indicating connection closure, we atomically remove the hashmap entry and notify the control-plane.

3.4 Flow Scheduling

FlexTOE leverages a work-conserving flow scheduler on the NIC data-path. The flow scheduler obeys transmission rate-limits and windows configured by the control-plane's congestion control policy. For each connection, the flow scheduler keeps track of how much data is available for transmission and the configured rate. Transmission rates and windows

are stored in NIC memory and are directly updated by the control-plane using MMIO.

We implement our flow scheduler based on Carousel [53]. Carousel schedules a large number of flows using a time wheel. Based on the next transmission time, as computed from rate limits and windows, we enqueue flows into corresponding slots in the time wheel. As the time slot deadline passes, the flow scheduler schedules each flow in the slot for transmission (§3.1.2). To conserve work, the flow scheduler only adds flows with a non-zero transmit window into the time wheel and bypasses the rate limiter for uncongested flows. These flows are scheduled round-robin.

4 Agilio-CX40 Implementation

This section describes FlexTOE's Agilio-CX40 implementation. Due to space constraints, the x86 and BlueField ports are described in detail in §E. FlexTOE's design across the different ports is identical. We do not merge or split any of the fine-grained modules or reorganize the pipeline across ports.

FlexTOE is implemented in 18,008 lines of C code (LoC). The offloaded data-path comprises 5,801 lines of C code. We implement parts of the data-path in assembly for performance. libTOE contains 4,620 lines of C, whereas the control path contains 5,549 lines of C. libTOE and the control plane are adapted from TAS. We use the NFP compiler toolchain version 6.1.0.1 for SmartNIC development.

Driver. We develop a Linux FlexTOE driver based on the `igb_uio` driver that enables libTOE and the control plane to perform MMIO to the SmartNIC from user space. The driver supports MSI-X based interrupts. The control-plane registers an `eventfd` for each application context in the driver. The interrupt handler in the driver pings the corresponding `eventfd` when an interrupt is received from the data-path for the application context. This enables libTOE to sleep when waiting for IO and reduces the host CPU overhead of polling.

Host memory mapping. To simplify virtual to physical address translation for DMA operations, we allocate physically contiguous host memory using 1G hugepages. The control-plane maps a pool of 1G hugepages at startup and allocates socket buffers and context queues out of this pool. In the future, we can use the IOMMU to eliminate the requirement of physically contiguous memory for FlexTOE buffers.

Context queues. Context queues use shared memory on the host, but communication between SmartNIC and host requires PCIe. We use scalable and efficient PCIe communication techniques [44] that poll on host memory locations when executing in the host and on NIC-internal memory when executing on the NIC. The NIC is notified of new queue entries via MMIO to a NIC doorbell. The context queue manager notifies applications through MSI-X interrupts, converted by the driver to an `eventfd`, after a queue has been inactive.

4.1 Near-memory Processing

An order of magnitude difference exists in the access latencies of different memory levels of the NFP-4000. For performance, it is critical to maximize access to local memory. The NFP-4000 also provides certain near-memory acceleration, including a lookup engine exposing a content addressable memory (CAM) and a hash table for fast matching, a queue memory engine exposing concurrent data structures such as linked lists, ring buffers, journals, and work-stealing queues. Finally, synchronization primitives such as ticket locks and inter-FPC signaling are exposed to coordinate threads and to sequence packets. We build specialized caches at multiple levels in the different pipeline stages using these primitives. Other NICs have similar accelerators.

Caching. We use each FPC's CAM to build 16-entry fully-associative local memory caches that evict entries based on LRU. The protocol stage adds a 512-entry direct-mapped second-level cache in CLS. Across four islands, we can accommodate up to 2K flows in this cache. The final level of memory is in EMEM. When an FPC processes a segment, it fetches the relevant state into its local memory either from CLS or from EMEM, evicting other cache entries as necessary. We allocate connection identifiers in such a way that we minimize collisions on the direct-mapped CLS cache.

Active connection database. To facilitate connection index lookup in the pre-processing stage, we employ the hardware lookup capability of IMEM to maintain a database of active connections. CAM is used to resolve hash collisions. The pre-processor computes a CRC-32 hash on a segment's 4-tuple to locate the connection index using the lookup engine. The pre-processor caches up to 128 lookup entries in its local memory via a direct-mapped cache on the hash value.

FPC mapping. FlexTOE's pipeline fully leverages the Agilio CX40 and is extensible to further FPCs, e.g. of the Agilio LX [40]. For island-local interactions among modules, we use CLS ring buffers. CLS supports the fastest intra-island producer-consumer mechanisms. Among islands, we rely on work-queues in IMEM and EMEM.

We use all but one general-purpose islands for the first three stages of the data-path pipeline (*protocol islands*). Each island manages a *flow-group*. While protocol and post-processing FPCs are local to a flow-group, pre-processors handle segments for any flow. We assign 4 FPCs to pre-/post-processing stages in each flow-group. Each island retains 3 unassigned FPCs that can run additional data-path modules (§5.1).

On the remaining general-purpose island (called *service island*), we host remaining pipeline stages and adjacent modules, such as context queue FPCs, the flow scheduler (SCH), and DMA managers. DMA managers are replicated to hide PCIe latencies. The number of FPCs assigned to each functionality is determined such that no functionality may become a

bottleneck. Sequencing and reordering FPCs are located on a further island with miscellaneous functionality.

Flow scheduler. We implement Carousel using hardware queues in EMEM. Each slot is allocated a hardware queue. To add a flow to the time wheel, we enqueue it on the queue associated with the time slot. Note that the order of flows within a particular slot is not preserved. EMEM support for a large number of hardware queues enables us to efficiently implement a time wheel with a small slot granularity and large horizon to achieve high-fidelity congestion control. Converting transmission rates to deadlines requires division, which is not supported on the NFP-4000. Thus, the control-plane computes transmission intervals in cycles/byte units from rates and programs them to NIC memory. This enables the flow scheduler to compute the time slot using only multiplication.

5 Evaluation

We answer the following evaluation questions:

- **Flexible offload.** Can flexible offload improve throughput, latency, and scalability of data center applications? Can we implement common data center features? (§5.1)
- **RPCs.** How does FlexTOE's data-path parallelism enable TCP offload for demanding RPCs? Do these benefits generalize across hardware architectures? Does FlexTOE provide low latency for short RPCs? Does FlexTOE provide high throughput for long RPCs? To how many simultaneous connections can FlexTOE scale? (§5.2)
- **Robustness.** How does FlexTOE perform under loss and congestion? Does it provide connection-fairness? (§5.3)

Testbed cluster. Our evaluation setup consists of two 20-core Intel Xeon Gold 6138 @ 2 GHz machines, with 40 GB RAM and 48 MB aggregate cache. Both machines are equipped with Netronome Agilio CX40 40 Gbps (single port), Chelsio Terminator T62100-LP-CR 100 Gbps and Intel XL710 40 Gbps NICs. We use one of the machines as a server, the other as a client. As additional clients, we also use two 2×18-core Intel Xeon Gold 6154 @ 3 GHz systems with 90 MB aggregate cache and two 4-core Intel Xeon E3-1230 v5 @ 3.4 GHz systems with 9 MB aggregate cache. The Xeon Gold machines are equipped with Mellanox ConnectX-5 MT27800 100 Gbps NICs, whereas the Xeon E3 machines have 82599ES 10 Gbps NICs. The machines are connected to a 100 Gbps Ethernet switch.

Baseline. We compare FlexTOE performance against the Linux TCP stack, Chelsio's kernel-based TOE[3], and the TAS kernel-bypass stack[4]. TAS does not perform well with the Agilio CX40 due to a slow NIC DPDK driver. We run TAS on the Intel XL710 NIC, as in [19], unless mentioned otherwise. We use identical application binaries across all baselines. DCTCP is our default congestion control policy.

---

[3]Chelsio does not support kernel-bypass.
[4]TAS [19] performs better than mTCP [17] on all of our benchmarks. Hence, we omit a comparison to mTCP and AccelTCP [37], which uses mTCP.
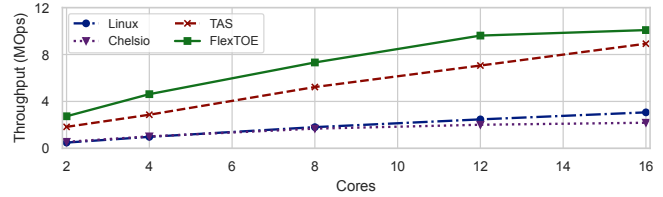
Figure 8. Memcached throughput scalability.
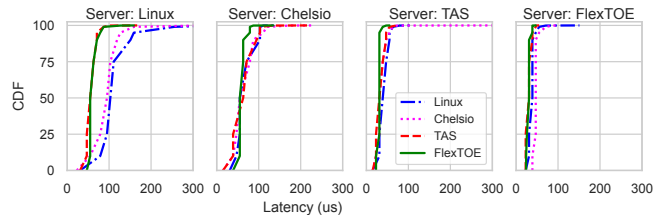
Figure 9. Latency of different server-client combinations.

5.1 Benefit of Flexible Offload

Application throughput scalability. Offloaded CPU cycles may be used for application work. We quantify these benefits by running a Memcached server, as in §2.1, varying the number of server cores. Figure 8 shows that, by saving host CPU cycles (cf. Table 1), FlexTOE achieves up to 1.6× TAS, 4.9× Chelsio, and 5.5× Linux throughput. FlexTOE and TAS scale similarly—both use per-core context queues. The Agilio CX becomes a compute-bottleneck at 12 host cores. Linux and Chelsio are slow for this workload, due to system call overheads, and do not scale well due to in-kernel locks.

Low (tail) latency. We repeat a single-threaded version of the same Memcached benchmark for all server-client network stack combinations. Latency distributions are shown in Figure 9. We can see that FlexTOE consistently provides the lowest median and tail Memcached operation latency across all stack combinations. Offload provides excellent performance isolation by physically separating the TCP data-path, even though FlexTOE's pipelining increases minimum latency in some cases (cf. §5.2).

Flexibility. Unlike fixed offloads and in-kernel stacks, FlexTOE provides full user-space programmability via a module API, simplifying development. Customizing FlexTOE is simple and does not require a system reboot. For example, we have developed logging, statistics, and profiling capabilities that can be turned on only when necessary. We make use of these capabilities during development and optimization of FlexTOE. We implemented up to 48 different tracepoints (including examples from bpftrace [13]) in the data-path pipeline, tracking transport events such as per-connection drops, out-of-order packets and retransmissions, inter-module queue occupancies, and critical section lengths in the protocol module for various event types. Table 2 shows that profiling degrades data-path performance versus the baseline by up to 24% when all 48 tracepoints are enabled. We also implement tcpdump-style traffic logging, including packet filters based on header

Build	Throughput (MOps)
Baseline FlexTOE	11.35
Statistics and profiling	8.67
`tcpdump` (no filter)	6.52
XDP (null)	10.87
XDP (vlan-strip)	10.83

Table 2. Performance with flexible extensions.
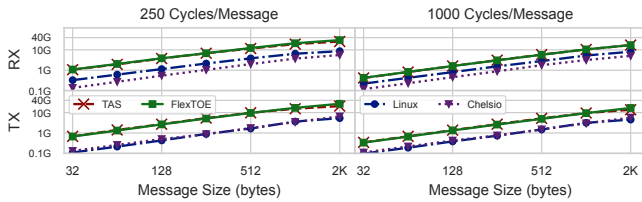
Figure 10. RPC throughput for saturated server.

fields. Logging naturally has high overhead (up to 43% when logging all packets). FlexTOE provides the flexibility to implement these features and to turn them on only when necessary.

Furthermore, new data-plane functionality leveraging the XDP API may be dynamically loaded into FlexTOE as eBPF programs. eBPF programs can be compiled to NFP assembly. This level of dynamic flexibility is hard to achieve with an FPGA as it requires instruction set programmability (overlays [52]). We measure the overhead of FlexTOE XDP support by running a null program that simply passes on every packet without modification. We observe only 4% decline in throughput. Common XDP modules, such as stripping VLAN tags on ingress packets, also have negligible overhead. Finally, connection splicing (cf. Listing 1 in the appendix) achieves a maximum splicing performance of 6.4 million packets per second, enough to saturate the NIC line rate with MTU-sized packets, leveraging only idle FPCs[5].

5.2 Remote Procedure Calls (RPCs)

RPCs are an important but difficult workload for flexible offload. Latency and client scalability requirements favor fast processing engines with large caches, such as found in CPUs and ASICs. Neither are available in on-path SmartNICs. We show that flexible offload can be competitive with state-of-the-art designs. We then show that FlexTOE's data-path parallelism is necessary to provide the necessary performance.

Typical RX / TX performance. We start with a typical server scenario, processing RPCs of many (128) connections, produced in an open loop by multiple (16) clients (multiple pipelined RPCs per connection). To simulate application processing, our server waits for an artificial delay of 250 or 1,000 cycles for each RPC. We run single-threaded to avoid the network being a bottleneck. We quantify RX and TX throughput separately, by switching RPC consumer and producer roles among clients and servers, over different RPC sizes.

---

[5]We are compute-limited by our Agilio CX. Using an Agilio LX, like AccelTCP, would allow us to achieve even higher throughput.
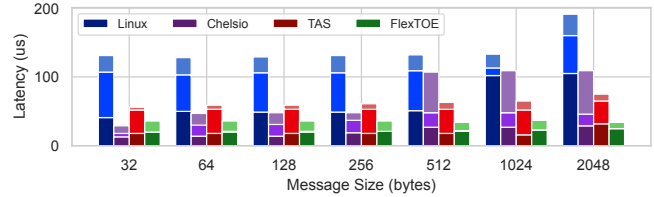
Figure 11. Median, 99p and 99.99p RPC RTT.

Figure 10 shows the results. For 250 cycles of processing overhead, FlexTOE provides up to 4× better throughput than Linux and 5.3× better throughput than Chelsio when receiving. For 2 KB message size, both TAS and FlexTOE reach 40 Gbps line rate, whereas Linux and Chelsio barely reach 10 Gbps and 7 Gbps, respectively. When sending packets, the difference in performance between Linux and FlexTOE is starker. FlexTOE shows over 7.6× higher throughput over both Linux and Chelsio for all message sizes. The gains remain at over 2.2× as we go to 1,000 cycles/RPC. Performance of TAS and FlexTOE track closely for all message sizes. This is expected as the single application server core is saturated by both network stacks (TAS runs on additional host cores).

We break down this result by studying the performance sensitivity of each TCP stack, varying each RPC parameter within its sensitive dynamic range. For these benchmarks, we evaluate the raw performance of the stacks, without application processing delays.

RPC latency. A client establishes a single connection to the server and measures single RPC RTT. Figure 11 shows the median and tail RTT for various small message sizes (stacked bars). The inefficiency of in-kernel networking is reflected in the median latency of Linux, which is at least 5× worse compared to other stacks. For message sizes < 256 B, Flex-TOE's median latency (20 us) is 1.4× Chelsio's median latency (14 us) and 1.25× TAS's median latency (16 us). FlexTOE's data-path pipeline across many wimpy FPCs increases median latency for single RPCs. However, FlexTOE has an up to 3.2× smaller tail compared to Chelsio and nearly constant per-segment overhead as the RPC size increases. In case of a 2 KB RPC (larger than the TCP maximum segment size), FlexTOE's latency distribution remains nearly unchanged. FlexTOE's fine-grain parallelism is able to hide the processing overhead of multiple segments, providing 22% lower median and 50% lower tail latency than TAS.

Per-connection throughput. In this setup, a client transfers a large RPC message to the server. In the first case (Figure 12a), the server responds with a 32 B response whereas in the second case (b), the server echoes the message back to the client (TAS performance is unstable with messages > 2 MB in this case—we omit these results). In the short-response case, Chelsio performs 20% better than the other stacks—Chelsio is a 100 Gbps NIC optimized for unidirectional streaming. However, it has 20% lower throughput as compared to FlexTOE in
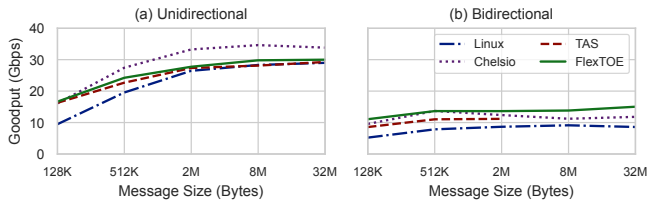
Figure 12. Large RPC throughput with varying RPC size.
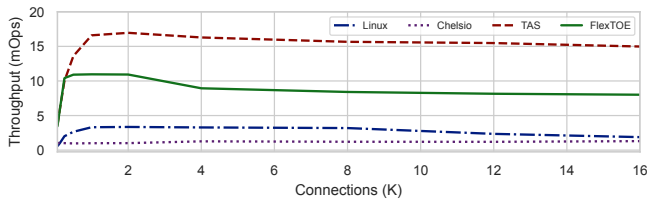
Figure 13. Connection scalability benchmark.

the echo case. Other stacks cannot parallelize per-connection processing, leading to limited throughput[6], while FlexTOE's throughput is limited by its protocol stage. FlexTOE currently acknowledges every incoming packet. For bidirectional flows, this quadruples the number of packets processed per second. Implementing delayed ACKs would improve FlexTOE's performance further for large flows.

Connection scalability. We establish an increasing number of RPC client connections from all 5 client machines to a multi-threaded echo server. To stress TCP processing, each connection leaves a single 64 B RPC in-flight. Figure 13 shows the throughput as we vary the number of connections. This workload is very challenging for FlexTOE as it exhausts fast memory and prevents per-connection batching, causing a cache miss at every pipeline stage for every segment. Up to 2K connections, FlexTOE shows a throughput of 3.3× Linux. TAS performs 1.5× better than FlexTOE for this workload. Flex-TOE is compute-bottlenecked[7] at the protocol stage, which uses 8 FPCs in this benchmark. Agilio CX caches 2K connections in CLS memory. Beyond this, the protocol stage must move state among local memory, CLS, and EMEM. EMEM's SRAM cache is increasingly strained as the number of connections increases. FlexTOE's throughput declines by 24% as we hit 8k connections and plateaus beyond that[8]. TAS's fast-path exhibits better connection scalability, as it has access to the larger host CPU cache, while Linux's throughput declines significantly. Chelsio has poor performance for this workload, as epoll() overhead dominates.

Benefit of data-path parallelism. To break down the impact of FlexTOE's data-parallel design on RPC performance,

---

[6]With multiple unidirectional flows, all stacks achieve line rate (Figure 15b).
[7]We expect that running FlexTOE on the Agilio LX with 1.2 GHz FPCs—1.5× faster than Agilio CX—would boost the peak throughput to match TAS performance. Agilio LX also doubles the number of FPCs and islands. It would allow us to exploit more parallelism and cache more connections.
[8]While we evaluate up to 16K connections, FlexTOE can leverage the 2 GB on-board DRAM to scale to 1M+ connections.

Design	Throughput (Mbps)	×	Latency (us) 50p	99.99p
Baseline	79.32	1	1,179	6,929
+ Pipelining	3,640.49	46	183	684
+ Intra-FPC parallelism	8,194.34	103	128	148
+ Replicated pre/post	11,086.93	140	94	106
+ Flow-group islands	22,684.69	286	46	58

Table 3. FlexTOE data-path parallelism breakdown.
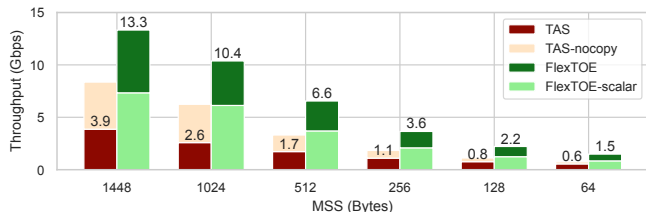
Figure 14. FlexTOE benefits on BlueField SmartNIC.

we repeat the echo benchmark with 64 connections, with each connection leaving a single 2 KB RPC in-flight (to be able to evaluate both intra and inter connection parallelism). Table 3 shows the performance impact as we progressively add data-path parallelism. Our baseline runs the entire TCP processing to completion on the SmartNIC before processing the next segment. Pipelining improves performance by 46× over the baseline. As we enable 8 threads on the FPCs (2.25× gain), we hide the latency of memory operations and improve FPC utilization. Next, we replicate the pre-processing and post-processing stages, leveraging sequencing and reordering for correctness, to extract 1.35× improvement and finally, with four flow-group islands, we see a further 2× improvement. We can see that each level of data-path parallelism is necessary, improving RPC throughput and latency by up to 286×.

Do these benefits generalize? We investigate whether data-path parallelism provides benefits across platforms. In particular, we investigate single connection throughput of pipelined RPCs across a range of maximum segment sizes (MSS) on a Mellanox BlueField [31] MBF1M332A-ASCAT 25 Gbps SmartNIC and on a 32-core AMD 7452 @ 2.35 GHz host with 128 GB RAM, 148 MB aggregate cache, and a conventional 100 Gbps ConnectX-5 NIC. We use a single-threaded RPC sink application, running on the same platform[9]. We compare TAS's core-per-connection processing to FlexTOE's data-parallelism. We replicate each of FlexTOE's pre and post processing stages 2×, resulting in 9 FlexTOE cores. Further gains may be achievable by more replication. To break down FlexTOE's benefits, we also compare to a FlexTOE pipeline without replicated stages (FlexTOE-scalar), using 7 cores.

Figure 14 shows BlueField results. FlexTOE outperforms TAS by up to 4× on BlueField (and 2.4× on x86). Depending on RPC size, FlexTOE accelerates different stages of the TCP data path. For large RPCs, FlexTOE accelerates data copy to

---

[9]BlueField is an off-path SmartNIC that is not optimized for packet processing offload to host-side applications (§2.3).
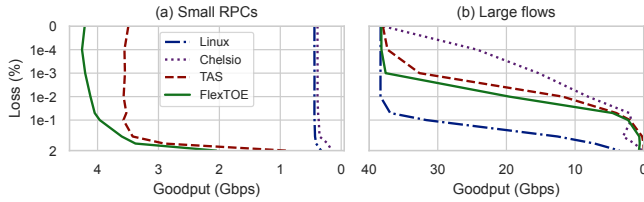
Figure 15. Throughput, varying packet loss rate.

Figure 16. Throughput distribution at line rate.

deg.	# con.	Tpt. (G)		Lat. 99.99p (ms)		JFI	
		on	off	on	off	on	off
4	16	9.51	9.47	5.98	11.58	0.98	0.95
4	64	9.51	9.23	10.75	44.39	0.96	0.73
4	128	9.48	8.96	13.74	64.25	0.99	0.53
10	10	3.66	1.04	2.50	18.26	0.95	0.78
20	20	1.76	0.36	7.35	138.32	0.95	0.46

Table 4. FlexTOE congestion control under incast.

socket payload buffers. To show this, we eliminate the step in TAS (TAS-nocopy), allowing TAS to perform at 0.5× FlexTOE on BlueField (and identical to FlexTOE on x86). For smaller RPCs, TAS-nocopy benefits diminish and FlexTOE supports processing higher packet rates. FlexTOE-scalar achieves only up to 2.3× speedup over TAS on BlueField (and 1.47× on x86), showing that only part of the benefit comes from pipelining. Finally, FlexTOE speedup is greater on the wimpier BlueField, resembling our target architecture (§2.3), than on x86. To save powerful x86 cores, some stages may be collapsed, even dynamically (cf. Snap [29]), at little performance cost.

5.3 Robustness

Packet loss. We artificially induce packet losses in the network by randomly dropping packets at the switch with a fixed probability. We measure the throughput between two machines for 100 flows running 64 B echo-benchmark as we vary the loss probability, shown in Figure 15a. We configure the clients to pipeline up to 8 requests on each connection to trigger out-of-order processing when packets are lost. Flex-TOE's throughput at 2% losses is at least twice as good as TAS and an order of magnitude better than the other stacks for this case. We repeat the unidirectional large RPC benchmark with 8 connections and measure the throughput as we increase the packet loss rate. For this case (b), Chelsio has a very steep decline in throughput even with 10^{-4}% loss probability. Linux is able to withstand higher loss rates as it implements more sophisticated reassembly and recovery algorithms, including selective acknowledgments—FlexTOE and TAS implement single out-of-order interval tracking on the receiver-side and go-back-n recovery on the sender. FlexTOE's behavior under loss is still better than TAS. FlexTOE processes acknowledgments on the NIC, triggering retransmissions sooner, and its predictable latency, even under load, helps FlexTOE recover faster from packet loss. We note that RDMA tolerates up to 0.1% losses [35], while eRPC falters at 0.01% loss rate [18]. Unlike FlexTOE, RDMA discards all out-of-order packets on the receiver side [35]. TAS [19] provides further evaluation of the benefits of receiver out-of-order interval tracking.

Fairness. To show scalability of FlexTOE's SCH (§3.4), we measure the distribution of connection throughputs of bulk flows between two nodes at line rate for 60 seconds. Figure 16 shows the median and 1st percentile throughput of FlexTOE and Linux as we vary the number of connections. For FlexTOE, the median closely tracks the fair share throughput and the tail

is 0.67× of the median. Linux's fairness is significantly affected beyond 256 connections. Jain's fairness index (JFI) drops to 0.36 at 2K connections for Linux, while FlexTOE achieves 0.98. Above 1K connections, Linux' median throughput is worse than FlexTOE's 1st percentile.

Incast. We simulate incast by enabling traffic shaping on the switch to restrict port bandwidth to various incast degrees and we configure WRED to perform tail drops when the switch buffer is exhausted. In this experiment, the client transfers 64 KB RPCs and the server responds with a 32 B response on each connection. As shown in Table 4, control-plane-driven congestion control in FlexTOE is able to achieve the shaped line rate, maintain low tail latency, and ensure fairness among flows under congestion. Disabling it causes excessive drops, inflating tail latency by 18.8× and skewing fairness by 2×.

6 Conclusion

FlexTOE is a flexible, yet high-performance TCP offload engine to SmartNICs. FlexTOE leverages fine-grained parallelization of the TCP data-path and segment reordering for high performance on wimpy SmartNIC architecture, while remaining flexible via a modular design. We compare FlexTOE to Linux, the TAS software TCP accelerator, and the Chelsio Terminator TOE. We find that Memcached scales up to 38% better on FlexTOE versus TAS, while saving up to 81% host CPU cycles versus Chelsio. FlexTOE provides competitive performance for RPCs, even with wimpy SmartNICs, and is robust under adverse operating conditions. FlexTOE's API supports XDP programs written in eBPF. It allows us to implement popular data center transport features, such as TCP tracing, packet filtering and capture, VLAN stripping, flow classification, firewalling, and connection splicing.

References

[1] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). In *Proceedings of the 2010 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '10, pages 63–74, New York, NY, USA, 2010. Association for Computing Machinery.

[2] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling programmable transport protocols in high-speed NICs. In *Proceedings of the 17th USENIX Conference on Networked Systems Design and Implementation*, NSDI '20, pages 93–110, USA, 2020. USENIX Association.

[3] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI '14, pages 49–65, USA, 2014. USENIX Association.

[4] Broadcom. Broadcom Stingray SmartNICs. https://www.broadcom.com/products/ethernet-connectivity/smartnic/ps225, 2018.

[5] Cavium. Cavium OCTEON Development Kits. https://cavium.com/octeon-software-develop-kit.html, 2018.

[6] Chelsio Communications. T6 ASIC: High performance, dual port unified wire 1/10/25/40/50/100Gb Ethernet controller. https://www.chelsio.com/wp-content/uploads/resources/Chelsio-Terminator-6-Brief.pdf, 2017.

[7] Andy Currid. TCP offload to the rescue: Getting a toehold on TCP offload engines—and why we need them. *Queue*, 2(3):58–65, May 2004.

[8] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. RouteBricks: Exploiting parallelism to scale software routers. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, SOSP '09, pages 15–28, New York, NY, USA, 2009. Association for Computing Machinery.

[9] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure accelerated networking: SmartNICs in the public cloud. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI '18, pages 51–64, USA, 2018. USENIX Association.

[10] Michael Galles and Francis Matus. Pensando distributed services architecture. *IEEE Micro*, 41(2):43–49, 2021.

[11] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over commodity ethernet at scale. In *Proceedings of the 2016 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '16, pages 202–215, New York, NY, USA, 2016. Association for Computing Machinery.

[12] Intel Corporation. Intel 82599 10 GbE controller datasheet. Revision 3.4, November 2019. https://www.intel.com/content/www/us/en/ethernet-controllers/82599-10-gbe-controller-datasheet.html.

[13] IO Visor Project, Linux Foundation. bpftace: High-level tracing language for Linux eBPF. https://github.com/iovisor/bpftrace, 2021.

[14] IO Visor Project, Linux Foundation. XDP: express data path. https://www.iovisor.org/technology/xdp, 2021.

[15] Jakub Kicinski and Nicolaas Viljoen, Netronome Systems. ebpf hardware offload to smartnics: cls bpf and xdp. https://www.netronome.com/media/documents/eBPF_HW_OFFLOAD_HNiMne8_2_.pdf, 2021.

[16] Jakub Kicinski and Nicolaas Viljoen, Netronome Systems. Xdp hardware offload: Current work, debugging and edge cases. https://www.netronome.com/media/documents/viljoen-xdpoffload-talk_2.pdf, 2021.

[17] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: A highly scalable user-level TCP stack for multicore systems. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI '14, pages 489–502, USA, 2014. USENIX Association.

[18] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Datacenter RPCs can be general and fast. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI '19, pages 1–16, USA, 2019. USENIX Association.

[19] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP acceleration as an OS service. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[20] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. LineFS: Efficient SmartNIC offload of a distributed file system with pipeline parallelism. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles*, SOSP '21, pages 756–771, New York, NY, USA, 2021. Association for Computing Machinery.

[21] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: Making rpcs first-class datacenter citizens. In *Proceedings of the 2019 USENIX Annual Technical Conference*, USENIX ATC '19, pages 863–879, USA, 2019. USENIX Association.

[22] Nikita Lazarev, Shaojie Xiang, Neil Adit, Zhiru Zhang, and Christina Delimitrou. Dagger: Efficient and fast RPCs in cloud microservices with near-memory reconfigurable NICs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, pages 36–51, New York, NY, USA, 2021. Association for Computing Machinery.

[23] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. ClickNP: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '16, pages 1–14, New York, NY, USA, 2016. Association for Computing Machinery.

[24] Xiaofeng Lin, Yu Chen, Xiaodong Li, Junjie Mao, Jiaquan He, Wei Xu, and Yuanchun Shi. Scalable kernel TCP design and implementation for short-lived connections. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 339–352, New York, NY, USA, 2016. Association for Computing Machinery.

[25] Linux. bpf(2) — linux manual page. https://man7.org/linux/man-pages/man2/bpf.2.html, 2021.

[26] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto SmartNICs using iPipe. In *Proceedings of the 2019 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, pages 318–333, New York, NY, USA, 2019. Association for Computing Machinery.

[27] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. E3: Energy-efficient microservices on SmartNIC-accelerated servers. In *Proceedings of the 2019 USENIX Annual Technical Conference*, USENIX ATC '19, pages 363–378, USA, 2019. USENIX Association.

[28] David A. Maltz and Pravin Bhagwat. TCP splice application layer proxy performance. *Journal of High Speed Networks*, 8(3):225–240, January 2000.

[29] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin

Vahdat. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 399–413, New York, NY, USA, 2019. Association for Computing Machinery.

[30] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the 1993 USENIX Winter Conference*, USENIX '93, page 2, USA, 1993. USENIX Association.

[31] Mellanox. Mellanox BlueField Platforms. http://www.mellanox.com/related-docs/npu-multicore-processors/PB_BlueField_Ref_Platform.pdf, 2018.

[32] memcached. Memcached, 2020. https://memcached.org/.

[33] Microsoft. Information about the TCP Chimney offload, receive side scaling, and network direct memory access features in Windows Server 2008. https://docs.microsoft.com/en-US/troubleshoot/windows-server/networking/information-about-tcp-chimney-offload-rss-netdma-feature.

[34] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. TIMELY: RTT-based congestion control for the datacenter. In *Proceedings of the 2015 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '15, pages 537–550, New York, NY, USA, 2015. Association for Computing Machinery.

[35] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting network support for RDMA. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 313–326, New York, NY, USA, 2018. Association for Computing Machinery.

[36] Jeffrey C. Mogul. Tcp offload is a dumb idea whose time has come. In *Proceedings of the 9th USENIX Conference on Hot Topics in Operating Systems*, HotOS '03, page 5, USA, 2003. USENIX Association.

[37] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. AccelTCP: Accelerating network applications with stateful TCP offloading. In *Proceedings of the 17th USENIX Conference on Networked Systems Design and Implementation*, NSDI '20, pages 77–92, USA, 2020. USENIX Association.

[38] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. The Click modular router. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, SOSP '99, pages 217–231, New York, NY, USA, 1999. Association for Computing Machinery.

[39] Netronome. Netronome Agilio CX SmartNIC. https://www.netronome.com/products/agilio-cx/, 2018.

[40] Netronome. Netronome Agilio LX SmartNIC. https://www.netronome.com/products/agilio-lx/, 2018.

[41] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. Understanding PCIe performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 327–341, New York, NY, USA, 2018. Association for Computing Machinery.

[42] Zhixiong Niu, Hong Xu, Peng Cheng, Qiang Su, Yongqiang Xiong, Tao Wang, Dongsu Han, and Keith Winstein. NetKernel: Making network stack part of the virtualized infrastructure. In *Proceedings of the 2020 USENIX Annual Technical Conference*, USENIX ATC '20, USA, 2020. USENIX Association.

[43] Zhixiong Niu, Hong Xu, Dongsu Han, Peng Cheng, Yongqiang Xiong, Guo Chen, and Keith Winstein. Network stack as a service in the cloud. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets-XVI, pages 65–71, New York, NY, USA, 2017. Association for Computing Machinery.

[44] NVM Express Workgroup. NVM Express: Base specification. https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4a-2020.03.09-Ratified.pdf, 2020.

[45] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T. Morris. Improving network connection locality on multicore systems. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 337–350, New York, NY, USA, 2012. Association for Computing Machinery.

[46] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '14, pages 1–16, Broomfield, CO, October 2014. USENIX Association.

[47] I. Pratt and K. Fraser. Arsenic: a user-accessible gigabit ethernet interface. In *Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Society*, volume 1 of *INFOCOM '01*, pages 67–76 vol.1, 2001.

[48] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 13–24. IEEE Press, 2014.

[49] RDMA Consortium. Architectural specifications for RDMA over TCP/IP. http://www.rdmaconsortium.org/.

[50] Renato J. Recio, Paul R. Culley, Dave Garcia, Bernard Metzler, and Jeff Hilland. A Remote Direct Memory Access Protocol Specification. RFC 5040, October 2007.

[51] Redis Labs. memtier_benchmark: Load generation and bechmarking NoSQL key-value databases. https://github.com/RedisLabs/memtier_benchmark, 2020.

[52] Hugo Sadok, Zhipeng Zhao, Valerie Choung, Nirav Atre, Daniel S. Berger, James C. Hoe, Aurojit Panda, and Justine Sherry. We need kernel interposition over the network dataplane. In *Proceedings of the 2021 Workshop on Hot Topics in Operating Systems*, HotOS '21, pages 152–158, New York, NY, USA, 2021. Association for Computing Machinery.

[53] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Vinh The Lam, Carlo Contavalli, and Amin Vahdat. Carousel: Scalable traffic shaping at end hosts. In *Proceedings of the 2017 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 404–417, New York, NY, USA, 2017. Association for Computing Machinery.

[54] Pravin Shinde, Antoine Kaufmann, Timothy Roscoe, and Stefan Kaestle. We need to talk about NICs. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, HotOS '13, page 1, USA, 2013. USENIX Association.

[55] Pensando Systems. Pensando DSC-25 distributed services card. https://pensando.io/wp-content/uploads/2020/03/Pensando-DSC-25-Product-Brief.pdf, 2020.

[56] The Linux Foundation. toe. https://wiki.linuxfoundation.org/networking/toe.

Field	Bits	Description
Pre-processor (connection identification)—15B:		
peer_mac	48	Remote MAC address
peer_ip	32	Remote IP address
local|remote_port	32	TCP ports
flow_group	2	hash(4-tuple) % 4
Protocol (TCP state machine)—43B:		
rx|tx_pos	64	RX/TX buffer head
tx_avail	32	Bytes ready for TX
rx_avail	32	Available RX buffer space
remote_win	16	Remote receive window
tx_sent	32	Sent unack. TX bytes
seq	32	TCP seq. number
ack	32	TCP remote seq. number
ooo_start|len	64	Out-of-order interval
dupack_cnt	4	Duplicate ACK count
next_ts	32	Peer timestamp to echo
Post-processor (ctx queue, congestion control)—51B:		
opaque	64	App connection id
context	16	Context-queue id
rx|tx_base	128	RX/TX buffer base
rx|tx_size	64	RX/TX buffer size
cnt_ackb|ecnb	64	ACK'd and ECN bytes
cnt_fretx	8	Fast-retransmits count
rtt_est	32	RTT estimate
rate	32	TX rate

Table 5. Connection state partitions (total: 108B).

A TCP Connection State Partitioning

To enable fine-grained parallelism, we partition connection state across pipeline stages. Table 5 shows the per-connection state variables, grouped by pipeline stage. Pre-processor state contains connection identifiers (MAC, IP addresses; TCP port numbers). Protocol state contains TCP windows, sequence and acknowledgment numbers, and host payload buffer positions. Post-processor state contains host payload buffer and context queue locations, and data-path congestion control state. DMA and context queue stages are stateless.

In aggregate, each TCP connection has 108 bytes of state, allowing us to offload millions of connections to the SmartNIC. In particular, we can manage 16 connections per protocol FPC, 512 connections per flow-group, and 16K connections in the EMEM cache. Using all of EMEM, we can support up to 8M connections.

B Connection Splicing Implementation

We implement AccelTCP's connection splicing in 24 lines of eBPF code. Listing 1 shows the entire code.

C TAS TCP/IP Processing Breakdown

Table 6 shows a breakdown of the per-packet TCP/IP processing overheads (summarized as *TCP/IP stack* in Table 1) in TAS for the Memcached benchmark conducted in §2.1. For

```
BPF_MAP_HASH_DECLARE(splice_tbl, SPLICE_MAX_FLOWS, \
    sizeof(struct pkt_4tuple_t), sizeof(struct tcp_splice_t));

int bpf_xdp_prog(struct xdp_md* ctx)
{
  struct tcp_splice_t state;
  struct pkt_hdr_t    *hdr = BPF_XDP_ADDR(ctx->data);
  struct pkt_4tuple_t *key = &hdr->ip.src;

  // Filter non-IPv4/TCP segments to control-plane
  if (!segment_ipv4_tcp(hdr))
   return XDP_REDIRECT;

  // Connection Control: Segments with SYN, FIN, RST
  // Atomically remove map entry and forward to control-plane
  if (segment_tcp_ctrlflags(hdr)) {
    BPF_MAP_DELETE_ELEM(splice_tbl, key);
    return XDP_REDIRECT;
  }

  if (BPF_MAP_LOOKUP_ELEM(splice_tbl, key, &state) < 0)
    return XDP_PASS;    // Send to data-plane

  patch_headers(hdr, &state);
  return XDP_TX;    // Send out the MAC
}

void patch_headers(struct pkt_hdr_t *hdr,
            struct tcp_splice_t *state)
{
  hdr->eth.src   = hdr->eth.dst;
  hdr->eth.dst   = state->remote_mac;
  hdr->ip.src    = hdr->ip.dst;
  hdr->ip.dst    = state->remote_ip;
  hdr->tcp.sport = state->local_port;
  hdr->tcp.dport = state->remote_port;

  hdr->tcp.seq  += state->seq_delta;
  hdr->tcp.ack  += state->ack_delta;
}
```

Listing 1. Connection splicing with XDP in FlexTOE.

each request, TAS performs loss detection (and potentially recovery) that involves processing the incoming request segment, generating an acknowledgement for it, and additionally, processing the acknowledgement for the response segment, consuming 42% of the total per-packet processing cycles. TAS spends 9% of the total cycles to prepare the response TCP segment for transmission and an additional 12% to schedule flows

Function	Cycles	%
Segment generation	130	9
Loss detection (and recovery)	606	42
Payload transfer	10	1
Application notification	381	26
Flow scheduling	172	12
Miscellaneous	141	10
Total	1,440	100

Table 6. Breakdown of TCP/IP stack overheads in TAS.

based on the rate configured by the congestion control protocol. TAS spends 26% of per-packet cycles interacting with the application, to notify when a request is received, to admit a response for transmission, and to free the transmission buffer when it is acknowledged. For small request-response pairs (32B in this case), the payload copy overheads are negligible.

D Control Plane

FlexTOE's control plane is similar to that of existing approaches that separate control and data-plane activities, such as TAS [19]. Using it, we implement control-plane policies, such as congestion control, per-connection rate limits, per-application connection limits, and port partitioning among applications (cf. [52]). We briefly describe connection and congestion control in this appendix. Retransmissions are described in §3.1.1 and §3.1.3. TAS [19] provides further description and evaluation of the control plane (named "slow-path" in the TAS paper).

Connection control. Connection control involves complex control logic, such as ARP resolution, port and buffer allocation, and the TCP connection state machine. The data-path forwards control segments to the control-plane. The control-plane notifies libTOE of incoming connections on listening ports. If the application decides to accept() the connection, the control-plane finishes the TCP handshake, allocates host payload buffers and a unique connection index for the data-path. It then sets up connection state in the data-path at the index location. Similarly, libTOE forwards connect() calls to the control-plane, which establishes the connection. On shutdown(), the control-plane disables the connection and removes the corresponding data-path state.

Congestion control. FlexTOE provides a generic control-plane framework to implement different rate and window-based congestion control algorithms, akin to that in TAS [19]. The control-plane runs a loop over the set of active flows to compute a new transmission rate, periodically. The interval between each iteration of the loop is determined by the round-trip time (RTT) of each flow. In each iteration, the control-plane reads per-flow congestion control statistics from the data-path to calculate a new rate or window for the flow. The rate or window is then set in the data-path flow scheduler (§3.4) for enforcement. We also monitor retransmission timeouts in the control iteration. FlexTOE implements DCTCP [1] and TIMELY [34] in this way.

E FlexTOE x86 and BlueField Ports

We have ported the FlexTOE data-path to the x86 and Blue-Field platforms. FlexTOE's design across the different ports is identical. We do not merge or split any of the fine-grained modules or reorganize the pipeline across ports. FlexTOE's decomposition, pipeline parallelism, and per-stage replication all generalize across platforms. Both ports are also almost identical to the Agilio-CX40 implementation (cf. §4) and were completed within roughly 2 person-weeks, demonstrating the great development velocity of a software TCP offload engine. We describe the implementation differences of each port to the Agilio-CX40 version in this section.

Hardware cache management. The hardware-managed cache hierarchies of x86 and BlueField obviate the need for software-managed caching that was implemented on Agilio. Instead of leveraging near-memory processing acceleration of the NFP-4000 (cf. §4.1), our ports implement multi-core ring buffers, flow lookup and packet sequencers in software. The more powerful x86 and BlueField cores make up for the difference in performance.

Symmetric core mapping. Unlike the NFP-4000, where FPCs are organized into islands, cores on x86 and BlueField have mostly symmetric communication properties, so the assignment of modules to cores is arbitrary and the manual FPC mapping step is omitted. However, we note that core mapping may still be beneficial, for example to leverage shared caches and node locality on multi-socket x86 systems. Each instance of a module runs on its own core. Apart from the six fine-grained pipeline modules: *pre-processing*, *protocol*, *post-processing*, *DMA*, *context queue*, and *SCH* shown in Figure 3, the ports utilize an additional *netif* module to interface with DPDK NIC queues to receive and transmit packets. Therefore, FlexTOE-scalar uses 7 cores and the FlexTOE-2× configuration uses 2 additional cores to replicate the pre and post-processing stages for a total of 9 cores.

Context queues use only shared memory. Our x86 and BlueField ports currently only support applications running on the same platform as FlexTOE. Hence, context queues always use shared memory rather than DMA. The corresponding DMA pipeline stage executes the payload copies in software using shared memory, rather than leveraging a DMA engine.

Platform-specific parameters. The replication factor of each pipeline stage is platform dependent. Stage-specific microbenchmarks on each platform can determine it. Our generalization experiments (§5.2) are designed to show that FlexTOE's data-parallelism can improve single connection throughput. Hence, we configure only one instance of the FlexTOE data-path pipeline in these versions (no flow-group islands—we do not process multiple connections in these experiments). Each port's pipeline uses the same number of stages as the Agilio-CX40 version, but we set different replication factors for the pre and post processing stages on x86 and BlueField (no replication and 2× replication). We do not attempt to find the optimal replication factor for best performance nor compact stages to reduce wasted CPU cycles.

Swift: Adaptive Video Streaming with Layered Neural Codecs

Mallesham Dasari, Kumara Kahatapitiya, Samir R. Das, Aruna Balasubramanian, Dimitris Samaras
Stony Brook University

Abstract

Layered video coding compresses video segments into layers (additional code bits). Decoding with each additional layer improves video quality incrementally. This approach has potential for very fine-grained rate adaptation. However, layered coding has not seen much success in practice because of its cross-layer compression overheads and decoding latencies. We take a fresh new approach to layered video coding by exploiting recent advances in video coding using deep learning techniques. We develop `Swift`, an adaptive video streaming system that includes i) a layered encoder that learns to encode a video frame into layered codes by purely encoding *residuals* from previous layers without introducing any cross-layer compression overheads, ii) a decoder that can *fuse* together a subset of these codes (based on availability) and decode them all in one go, and, iii) an adaptive bit rate (ABR) protocol that synergistically adapts video quality based on available network and client-side compute capacity. `Swift` can be integrated easily in the current streaming ecosystem without any change to network protocols and applications by simply replacing the current codecs with the proposed layered neural video codec when appropriate GPU or similar accelerator functionality is available on the client side. Extensive evaluations reveal `Swift`'s multi-dimensional benefits over prior video streaming systems.

1 Introduction

Internet video delivery often encounters highly variable and unpredictable network conditions. Despite various advances made, delivering the highest possible video quality continues to be a challenging problem due to this uncertainty. The problem is more acute in wireless networks as the channel conditions and mobility adds to the uncertainty [39, 46]. Interestingly, the next generation wireless networks may even make the problem more challenging (e.g., 60GHz/mmWave [10, 11, 38]).

To counter the challenges posed by such varying network capacity, current video delivery solutions predominantly practice adaptive streaming (e.g., DASH [50]), where a source video is split into segments that are encoded at the server into multiple bitrates providing different video qualities, and a client runs an adaptive bitrate (ABR) algorithm to dynamically select the highest quality that fits within the estimated network capacity for the next segment to be downloaded.

Need for layered coding. Most of the current commercial ABR algorithms adopt a monolithic encoding practice (e.g.,
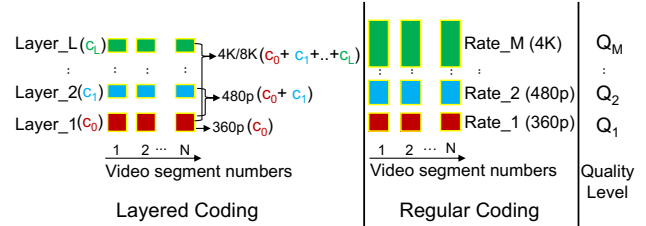
Figure 1: *Layered* vs. *Regular* coding methods. In Regular coding the video segments are coded independently at different qualities. In Layered coding a given quality can be reconstructed by combining codes for multiple layers thus facilitating incremental upgrades or downgrades.

H.265 [53]), where the same video segment is encoded 'independently' for each quality level. The decision on fetching a segment at a certain quality is considered *final* once the ABR algorithm makes a determination based on estimating the network capacity. However, these estimations are far from accurate, resulting in either underutilizing or overshooting the network capacity. For example, the ABR algorithm may fetch at a *low quality* by underestimating the network capacity, or it may fetch at a high quality causing *video stalls* by overestimating. Consequently, even the optimal ABR algorithms fail to provide a good quality of experience (QoE), as such rigid methods that do not fit the need of the streaming conditions.

An alternate technique, called *layered coding*, has been long studied [12, 14, 36, 47, 67] that can avoid the above streaming issues. The key idea here is that, instead of *independently* encoding the segment in different qualities, the segment is now encoded into *layers*; the base layer provides a certain video quality, and additional layers improve the video quality when applied over the base layer. See Figure 1. This means that, if the network throughput improves, one can fetch additional layers to improve video quality at a much lower cost compared to a *regular* codec.[1] We use the term *regular* coding to indicate the current practice of independent encoding in multiple qualities (current standards such as H.265/HEVC [53]).

Challenges with layered coding. Layered coding, however, faces two nontrivial challenges: *compression overhead*, and *coding latency*. The compression overhead mainly comes from not having the inter-layer frame prediction to avoid reconstruction drift in quality [29, 42, 61, 67]. On the other hand, the decoding latency is a function of the number of layers as

---

[1] We use terms coding or codec for encoding and decoding together. Also we use the terms encoding/compression, decoding/decompression interchangeably.

each layer needs to be decoded separately. Notwithstanding these issues, some studies have indeed applied layered coding in streaming and have shown slightly better QoE compared to the regular coding methods, benefiting from its ability to do dynamic quality upgrades [31]. However, they do not address either the overhead or the latency issues directly. Industry streaming solutions continue to adopt the regular codecs, shipping these codecs in hardware to avoid computational challenges, making it harder to adopt new innovations.

Neural video codecs. A learning approach to video coding has shown tremendous improvement in compression efficiency in just a few years [43, 60, 65]. Figure 2 shows bits-per-pixel vs PSNR plots[2] for several generations of codecs of two types – *neural codecs* that use deep learning and traditional *algorithmic* codecs that use the popular H.26x standards. It took algorithmic codecs 18 years to make the same progress that neural codecs achieved in the last 4 years! One reason for this rapid development is that neural codecs can run in software that can be integrated as part of the application, support agile codec development and provide royalty-free codecs. Further, they run on data parallel platforms such as GPUs that are increasingly available and powerful.

There are several insights in using neural codecs for video coding – **1)** unlike the traditional *layered* coding methods where it is nontrivial to handcraft each layer[3] to have unique *information*, a neural network's loss function can be optimized to encode a video frame into unique layered codes by purely encoding *residuals* from previous layers without introducing a reconstruction drift; **2)** a neural network can be trained to accept a subset of the layered codes and decode all of them in a single-shot, which again was traditionally difficult to do with a handcrafted algorithm due to nonlinear relationships among the codes. Additionally, **3)** neural codecs enable software-driven coding. We note here that GPUs or similar accelerators for neural network computation are critical for success with neural codecs. Fortunately, they are increasingly common in modern devices.

Swift. Based on the above insights, we present Swift, a novel video streaming system using *layered* coding built on the principles of neural video codecs [32, 60, 65].[4] We show that learning can address the challenges of layered coding mentioned earlier – there is no additional compression overhead with further layering and the decoding latency is independent of the number of layers. Swift consists of three design components: i) server-side encoder plus decoder, ii) client-side decoder, and iii) ABR protocol adapted to layered coding and varying compute capacity (in addition to varying network capacity).

---

[2]Bits-per-pixel captures compression efficiency and PSNR (peak signal-to-noise ratio) captures image quality. Both metrics together capture codec performance.

[3]Throughout the paper, the term 'layer' refers to compressed code layers, not neural network layers.

[4]The source code of Swift is available at the following site: https://github.com/VideoForage/swift.
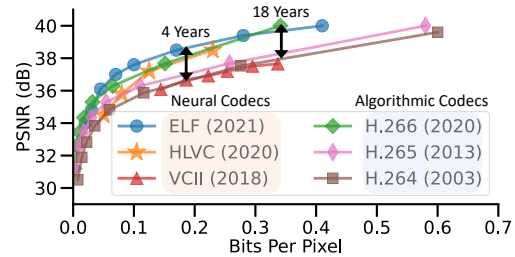
Figure 2: Evolution of neural and algorithmic video codecs showing compression efficiency plots across generations.

We evaluate Swift with diverse video content and FCC-released real-world network traces [8]. We compare Swift with state-the-art streaming algorithms that combine either regular coding [35, 51, 52] or layered coding [31] with state-of-the-art ABR algorithms. In terms of QoE, Swift outperforms the next-best streaming alternative by 45%. It does so using 16% less bandwidth and has a lower reaction time to changing network conditions. In terms of the neural codec, Swift's layered coding outperforms the state-of-the-art layered codec (SHVC [12]) by 58% in terms of compression ratio, and by $\times 4$ (for six layers) in terms of decoding latency. In summary, our contributions are the following:

- We show how deep learning-based coding can make layered coding both practical and high-performing, while addressing existing challenges that stymied the interest in layered coding.
- We design and build Swift to demonstrate a practical layered coding based video streaming system. Swift is an embodiment of deep learning-based encoding and decoding methods along with a purpose-built ABR protocol.
- We comprehensively evaluate and showcase the multi-dimensional benefits of Swift in terms of QoE, bandwidth usage, reaction times and compression efficiency.

2 Motivation

2.1 Limitations of Today's Video Streaming Due to Regular Coding

Today's video providers predominantly use source rate adaptation (e.g., MPEG-DASH [50]) where video segments are encoded at different qualities on the server and an adaptive bitrate (ABR) algorithm chooses the best quality segment to be downloaded based on the network capacity.

The streaming solutions that are widely deployed, use *regular*, standards-driven, algorithmic coding methods such as H.265/HEVC [53] or VP9 [4] for encoding video segments. These coding methods do not allow segments to be upgraded or downgraded based on network conditions.

Figure 3 illustrates this problem using an example experiment (more details about methodology are described in §6.1). The figure shows the quality of segments that are fetched by different state-of-the-art ABR algorithms that use regular
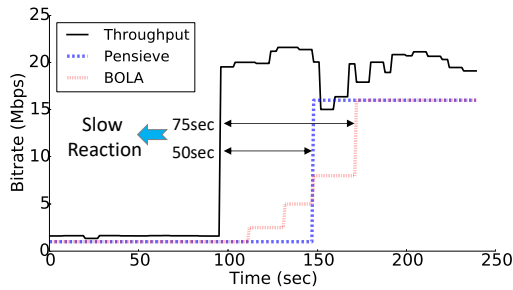
(a) Most ABR algorithms (BOLA, Penseive) cannot upgrade the quality of a video segment once downloaded and are slow to react to changing network conditions.
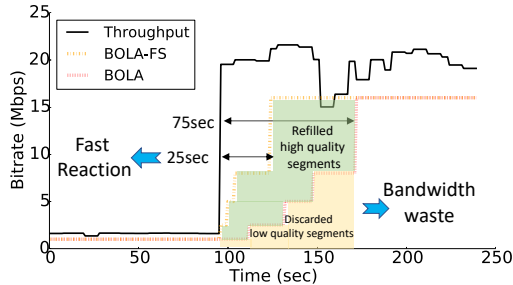
(b) BOLA-FS does allow video quality to be upgraded by re-downloading a higher quality segment. However, the previously downloaded segment is wasted.

Figure 3: Limitations of today's ABR algorithms because of regular coding: either slower reaction to network conditions or bandwidth wastage to achieve faster reaction time to highest quality. The reaction latency includes time to notice throughput increase as well as playing the buffered segments, and hence segment duration (5 sec here) plays a role. Penseive aggressively controls video quality fluctuations to compensate for incorrect bandwidth prediction, and hence the sudden jump in quality compared to BOLA.

coding. During the experiment, the throughput improves drastically at the 100 second mark. Two state-of-the-art streaming algorithms, Penseive [35] and BOLA [52], cannot upgrade the quality of a segment once the segment has been downloaded. This causes a slow reaction to adjust to the improved throughput. In Figure 3(b) however, BOLA-FS [51], a version of BOLA, does allow the higher quality segment to be re-downloaded when the network conditions improve. However, the previously downloaded lower quality segment is discarded, resulting in wasted bandwidth.

2.2 Layered Coding

A more suitable coding method to address the above issues is layered coding, where a video segment is encoded into a *base layer* (providing the lowest playback quality level) and multiple *enhancement layers* as shown in Figure 1. Clearly, layered coding gives much finer control on rate adaptation compared to regular coding. For example, multiple enhancement layers for the same segment can be fetched incrementally as the esti-
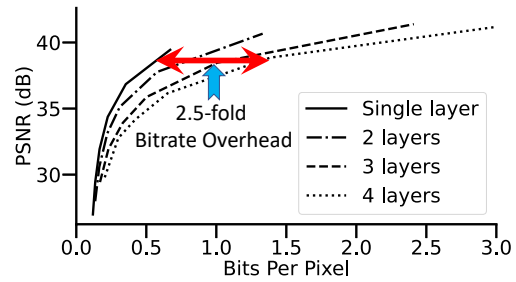
Figure 4: Compression efficiency of traditional layered coding. We use H.265 [53] and its layered extension SHVC [12] to encode the videos (described in §6.1). The single layer bitrate curve is same for both, and the additional layers are for SHVC. As shown, SHVC requires 2.5× more bits for 4 layers of SHVC compared to a single layer for the same quality.

mate of the network capacity improves closer to the playback time, which is not possible in case of regular coding.

2.3 Challenges of Adopting Traditional Layered Coding in Video Streaming

Layered coding has typically been developed and implemented as an extension to a *regular* coding technique. Published standards demonstrate this dependency: SHVC [12] has been developed as an extension of H.265 [53], similarly, older SVC [47] as an extension for H.264 [57]. Developing layered coding as an extension on top of a *regular* coding introduces multiple challenges in real-life deployments:

1) *Cross-layer compression overhead:* The key to large compression benefits in current generation video coding standards (e.g., ≈ 2000× compression ratio for H.265 [53]) is *inter*-frame prediction – the consecutive frames are similar and so it is efficient to simply encode the difference between consecutive frames. However, using the inter-layer frame prediction across enhancement layers of the current frame with respect to the previous frame makes video quality drift during decoding [29, 42, 61, 67]. To minimize or avoid the drift, most of the *layered* coding methods do not use inter-frame prediction across layers and thus lose out on its compression benefits [11, 17, 31]. In effect, to achieve the same quality, layered coding (e.g., SHVC) requires significantly more bits compared to its *regular* counterpart (e.g., H.265). In our study, we find that a 4-layer SHVC coding method needs 2.5× bits per pixel compared to its *regular* coding counterpart, H.265 (see Figure 4).

2) *High encoding and decoding latency:* The computational complexity of these algorithmic codecs mainly comes from the motion estimation process during inter-frame prediction [53, 57]. During the motion estimation, it is useful - for each pixel - to encode its motion vector, i.e., where its relative location was in the previous frame. The motion vectors are computed for each frame by dividing the frame into thousands of blocks of pixels and searching a similar block in the previous frames. In general, the codecs use a set of previous

frames to search blocks in each frame making it computationally expensive. The process becomes even more complex in case of layered coding because each layer has to be decoded one after the other because of the dependency of a layer on the previous one (to exploit the content redundancy) [11,27,30]. This serial process of layered coding makes the latency to be a function of number of layers, and therefore the latency increases progressively as we increase the number of layers.

Figure 5 shows per-frame decoding latency of the state-of-the-art layered coding (i.e., SHVC) of a 1-min video on a desktop with configuration described in §6.1. As shown, it takes more than 100ms to decode each frame for 5 layers, an order of magnitude increase in coding latency compared to its regular counterpart H.265 (an x265 [7] implementation).
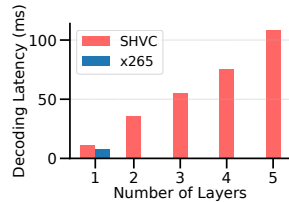
Figure 5: Latency challenges of traditional layered coding. The decoder is run on a high-end Desktop (as described in §5) using a single-threaded implementation of SHVC [3].

Despite several optimizations in the past, such range of latencies makes it infeasible to realize real-time decoding on heterogeneous platforms. Recent studies (e.g., Jigsaw [11]) tackle this challenge by proposing a lightweight *layered* coding method (using GPU implementation), but the latency is still a function of number of layers.

Because of these challenges, traditional layered coding is not used in practice today. In this work, rather than approaching this problem with yet another extension, we seek to explore layered coding via a clean-slate, learning-based approach with a goal towards efficient layered compression by embracing the opportunities of new hardware capabilities (e.g., GPUs and other data parallel accelerators).

2.4 Layered Coding using Neural Codecs

Video compression has recently experienced a paradigm shift in the computer vision community due to new advances in deep learning [32, 43, 60, 65]. The compression/decompression here is achieved using neural networks that we refer to as neural video codecs.

The basic idea is the use of an AutoEncoder (AE), a neural network architecture used to learn efficient encodings that has long been used for dimentionality reduction purposes [20]. The AE consists of an encoder and a decoder. The encoder converts an input video to a *code* vector that has a lower dimension than the input size, and the decoder reconstructs (perhaps with a small error) the original input video from the low-dimension *code* vector. The neural network weight parameters (W_i for encoder and W_i' for decoder) are trained by minimizing the *reconstruction error*, that is, minimizing the difference between the input and the output of the decoder. The smaller the code, the larger the compression factor but
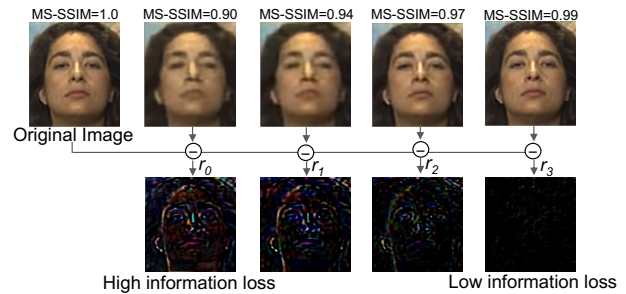
Figure 6: Illustrating the residuals ($r_0, \ldots, r_3$) from an original frame to a series of compressed-then-decoded frames. MS-SSIM [56] is a perceptual measure of image quality. A highly compressed frame (lowest MS-SSIM) has more residual information (r_0).

higher the reconstruction error.

Our insight in using Autonencoders is that a their loss function can be optimized to encode a video frame into unique layered codes by purely encoding residuals from previous layers, unlike the traditional layered coding where it is nontrivial to handcraft each layer to have unique information.

3 Swift

3.1 Overview

Autoencoders are already shown to provide similar or better performance relative to traditional codecs [32,43,60]. Recent work such as Elf-vc [43] is also able to use Autoencoders to provide flexible-rate video coding to fit a target network capacity or achieve a target compression quality. However, current work does not provide a way to encode in the video in incrementally decodable layers. To do this, we make use of *residuals* to form layered codes. A residual is the difference between the input to an encoder and output of the corresponding decoder. Residuals has been used in the past for tasks such as recognition and compression to improve the application's efficiency (e.g., classification accuracy or compression efficiency) [19,55,60].

Swift uses residuals for designing layered codecs for video streaming. The idea is to employ a chain of Autoencoders of identical structure. Each Autoencoder in the chain encodes the residual from the previous layer, with the very first Autoencoder in the chain (implementing the base layer) encoding the input video frames. Figure 6 shows an example, where the residuals are shown from an original frame to a series of progressively compressed-then-decoded frames. The first decoded frame (marked with $MS\_SSIM = 0.9$) has a relatively high loss from the original frame. As a result, the residual r_0 has more information. When this residual information is used for the next layer's compression, the resulting decoded frame is closer to the original, and in-turn the residual has less information, and so on.

The above 'iterative' chaining implicitly represents a layered encoding mechanism. Each iteration (i.e., layer) pro-

duces a compressed version of the original video that we call 'code.' These codes encode incremental information such that with more such codes decoded, the reconstruction becomes progressively closer to the original. `Swift` essentially uses this mechanism of residuals to create the layered codes. Such iterative minimization of residual also acts as an implicit regularization to guide the reconstruction (at a given bandwidth), instead of closely-following classical compression methods as in Elf-vc [43].

Figure 7 shows the Autoencoder architecture (more details in §3.2) on the server side. The architecture jointly learns *both* the encoder and decoder in each layer. As before, the Autoencoder's weight parameters are trained to minimize the reconstruction error between the input and output of the decoder. In this process, the encoder generates a compact code in each layer which is a compressed version of the input video frames. These codes are transmitted to the client, where they can be decoded for progressively better reconstructions.

The decoder learnt at the server is then optimized further (§3.3) to be used at the client side. The client decoder initially reconstructs the base layer from the first code. Then, if more layers/codes are downloaded from the server, the decoder reconstructs the residuals from the second layer onward, and combines with the previous reconstruction(s) to generate the output video frame.

Overall, `Swift` has three main components:

1. A learning-based layered encoder-decoder pair in a single neural network to create residual-based layered codes on the server-side (§3.2).
2. A separate learning-based decoder on the client side. This decoder can decode any combination of layered codes in a single-shot for real-time decoding (§3.3).
3. Extension of an ABR algorithm that can integrate the codec into a complete end-to-end system (§4).

3.2 Layered Neural Encoder

We first describe how the encoder and the decoder are trained at the server side. Assume, I^t is the image or video frame at time t, for $t \in \{0, 1, ...\}$. The encoder ($\mathcal{E}$) takes each of these frames as input and generates a compact *code vector* (c) for each frame, i.e., $c^t = \mathcal{E}(I^t)$. This code for each frame is constructed by exploiting the redundancy across multiple previous frames in the video. Therefore, the encoder takes a set of previous frames as reference in order to encode each frame. The decoder ($\mathcal{D}$) reconstructs the frame $\hat{I}^t$ given c^t, i.e., $\hat{I}^t = \mathcal{D}(c^t)$. The optimization problem here is to train $\mathcal{E}$ and $\mathcal{D}$ pairs so as to minimize the difference between $\hat{I}^t$ and I^t. Since we add our layered coding as a generic extension to any neural codec without changing its internal logic, $\mathcal{E}$ and $\mathcal{D}$ can be assumed as blackboxes. An example of a neural codec is presented in Appendix A.

Figure 7 shows the design of our layered encoder-decoder network on the server-side. Here, each iteration (or layer)
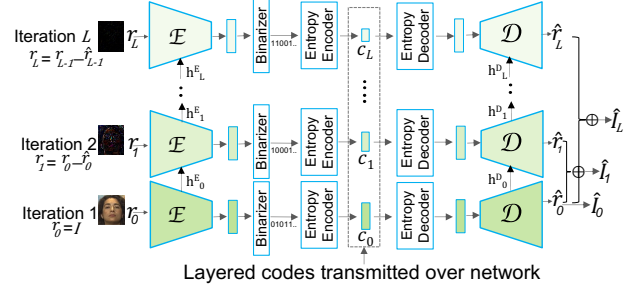
Figure 7: Deep learning based layered coding: a) iterative encoding ($\mathcal{E}$) and decoding ($\mathcal{D}$): in each iteration, $\mathcal{E}$ encodes a residual into a code (c_i) and the decoded output (from $\mathcal{D}$) is used to generate the residual for the next iteration.

encodes a residual r_i into a code c_i, where residual r_i is the difference between the encoder input and decoder output in the previous layers. For the very first iteration, the encoder directly encodes the the original video frame. Representing this mathematically: $c_i = \mathcal{E}(r_i)$ and $r_i = r_{i-1} - \hat{r}_{i-1}$ with $\hat{r}_i = \mathcal{D}(c_i)$, for $i = 0, \ldots, L$, with the exception that for $i = 0$ (base layer), $r_0 = I$.

At each iteration, the decoder can enhance the quality of the video frame with a plain arithmetic sum of the outputs of all previous iterations along with the base layer output. The key here is that both $\mathcal{E}$ & $\mathcal{D}$ have separate hidden states ($h^E *$ and $h^D *$) that get updated iteratively, sharing information between iterations. In fact, this subset of weights shared across iterations, allows better reconstruction of residuals. The entropy (i.e., the information) is very high in the initial layers, but progressively decreases due to the presence of the hidden connections and thus the code size becomes progressively smaller. The training objective for these iterative encoder-decoder pairs is to minimize the `L1` reconstruction loss for the residuals:

$$\mathcal{L}_{\text{rec}} = \frac{1}{L} \sum_{i=0}^{L-1} \|\mathcal{D}(c_i) - r_i\|_1$$

All Autoencoders $\mathcal{E}$ & $\mathcal{D}$ in the chain share the same network and thus have identical input and output sizes. They produce the same code sizes for all iterations. `Swift` relies on a separate entropy encoding stage (Figure 7) to create the residual codes that allocate proportional number of bits to match the entropy in each iteration. The fixed length code vector from the output of the encoder $\mathcal{E}$ is binarized and passed through a traditional entropy encoder similar to CABAC [54].

Note that the learned codec can work with a variety of input video resolutions, and hence we do not need to train a separate model for each video resolution. This is mainly because the Autoencoder here takes one or more video frames as input and extracts the features through convolutions (e.g., Conv2D [41]). Each convolutional kernel (with a fixed size of $k \times k$ pixels that is much smaller than the input resolution) is applied in a sliding window fashion on $k \times k$ blocks of pixels to reduce the dimensions and form the Autoencoder's

compact code vector. In our codec we use 4 downsampling convolution blocks to reduce the dimensions. This makes any input resolution ($w \times h$) to be downsampled to ($w/16) \times (h/16)$ resolution times the Autoencoder's bottleneck bits (b) after the encoding stage (see Appendix A). Therefore during the testing, the encoder's output for a 352×288 resolution would be $22 \times 18 \times b$, while it is $80 \times 45 \times b$ for 1280×720 resolution. Similarly the codec scales with other resolutions during testing.

3.3 Layered Neural Decoder

The above iterative coding design already includes the decoder (Figure 7) that can reconstruct the video from the layered codes. In principle, the client can use the same decoder already designed and learned on the server-side. However, the iterative method incurs decoding latency proportional to the number of iterations. This is because the residual codes are created separately in each iteration and the decoder cannot decode a code (c_i) unless the previous iteration of encoder encodes c_{i-1} and the corresponding decoder decodes it to form the residual r_i.

This latency is acceptable for video servers/CDNs that encode the videos offline and store them ready for on-demand streaming, but clients need to decode the video in real-time (≈ 30 fps). To address this, we develop a separate design of *single-shot* decoder to be used at the clients, that can take any combination of the codes as input and decode the corresponding frames in one shot. See Figure 8.

The codes available at the client are fused and padded with zeros up to a predetermined code length (corresponding to L levels) to account for unavailable codes. They are then fed into a 'multi-headed' decoder (H) as shown in the Figure 8. In each head, the padded version of each code c_i is separately processed through individual neural networks prior to combining them into a common network.

Figure 8: Single-shot decoder ($\mathcal{D}^{\text{ss}}$): a variable sized decoder that takes a subset of the codes to reconstruct a video frame in one go.

When higher layers are unavailable, the corresponding heads will have no effect on reconstruction, and when available, generate a desired residual mapping. Essentially, these multiple heads are lightweight and the common network (after combining the residual codes after multiple-heads) follows the same decoder architecture $\mathcal{D}$ from §3.2, but within one model. The heads or the common decoder do not share any parameters, in contrast to the iterative decoder which shares hidden states across iterations. Note that for preparing residual codes on the server-side, we

still need the iterative $\mathcal{E}$-$\mathcal{D}$ architecture as described in §3.2. To distinguish from the server-side decoder ($\mathcal{D}$), we denote this client-side single-shot decoder as $\mathcal{D}^{\text{ss}}$.

We train $\mathcal{D}^{\text{ss}}$ by extending the objective function used at the server-side. Specifically, in addition to the loss function at the server-side decoder ($\mathcal{D}$) which reconstructs a residual, we add a loss function that corresponds to the actual image reconstruction at client-side decoder ($\mathcal{D}^{\text{ss}}$) using the available code layers. Using L1 loss function, both the objectives are as shown below.

$$\mathcal{L}_{\text{rec}} = \frac{1}{L} \sum_{i=0}^{L-1} \Big[\underbrace{\big\| \mathcal{D}(c_i) - r_i \big\|_1}_{\text{residual quality loss}} + \underbrace{\big\| \mathcal{D}^{\text{ss}}(\oplus_{k=0}^{i} c_k) - I \big\|_1}_{\text{image quality loss}} \Big]$$

Here, the server-side decoder ($\mathcal{D}$) reconstructs the residual image r_i at each iteration based on the code c_i, while the client-side $\mathcal{D}^{\text{ss}}$ reconstructs the original image I based on the subset of the codes available at the corresponding iteration, i.e., $c_0 \ldots c_i$. This function allows us to train the encoder, and both the decoders (server and clients-side) in a single training loop. During the training, all three models $\mathcal{E}$, $\mathcal{D}$, and $\mathcal{D}^{\text{ss}}$ are jointly optimized by summing up the loss computed for $\mathcal{E}$ and $\mathcal{D}$ in §3.2 and the direct loss computed for $\mathcal{D}^{\text{ss}}$ that corresponds to original image reconstruction. This joint training with a more complex objective (i.e., multiple loss functions) does affect the performance of server-side decoder. The iterative decoder from §3.2 has simpler objective than $\mathcal{D}^{\text{ss}}$, and hence its performance is better when trained independently (in which case only the second term in the loss function is sufficient for $\mathcal{D}^{\text{ss}}$) compared to trained jointly. In our experiments we observe very little drop in quality on average with the joint training – that would be almost imperceptible to users. Moreover, training each of these models can also incur additional computation costs on servers.

4 Streaming with Layered Neural Codecs

Swift's layered neural codes introduces two challenges to end-to-end streaming. The first challenge arises because Swift's decoder at the client is expected to be run on the GPU or other similar data-parallel accelerators and run in software, instead of dedicated fixed hardware decoders as is the norm for regular codecs. Even though the software codecs have advantages in terms of on-demand codec upgrades and agile development, it raises the possibility of resource contention with other applications. Since GPU resource availability can vary [33, 45], the client needs to be able to adapt to the available resources.

The second challenge is in bitrate selection. Video streaming protocols encode each video segment into different qualities and uses ABR to select the next best quality video segment to stream. However, the ABR algorithm in Swift has a more complex choice—should one fetch the next segment at the highest possible quality or upgrade the current segment by
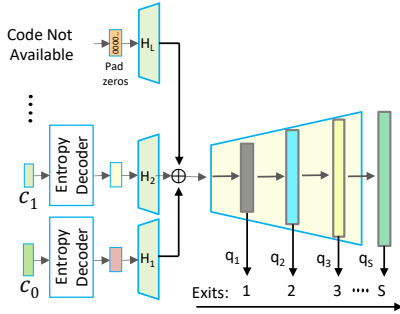
Figure 9: Scalable decoding using multiple exit heads to adapt to dynamic compute capacity. Each exit provides a different trade-off between compute capacity (and in-turn time required to decode) and quality of the video segment.

fetching additional layers? This question is made even more complex because the end-to-end streaming performance is affected both by the network and the compute variability at the client (see above). Traditional bitrate adaptation techniques are designed to only adapt to network variability.

In this section, we describe the design of a scalable decoder and neural-adapted ABR algorithm to tackle the challenges.

4.1 Scaling the Decoder based on Compute Capacity

The decoder architecture (shown in Figure 8) uses network with a certain depth. [5] As is common in Autoencoders, the more the depth, the better is the decoding accuracy, but lower the depth, lower is the compute requirement.

`Swift` exploits this trade-off by designing multiple, lightweight, output heads at different depths of the network. The decoder then operates at different design points in the accuracy vs compute requirement trade-off by exiting at different depths depending on the GPU capacity. We define GPU capacity as the percentage of time over the past sample period (1 sec in our case) during which one or more cores was executing on the GPU. For example, a 100% GPU utilization means all of the GPU cores are busy with other applications in the last sample period. To this end, we introduce a number of early-exit heads (hd_j, where $j = 1..S$) in the D^{ss} decoder that are corresponding to different output video qualities. See Figure 9. For example, if there are 5 exits in the network, then each exit depth outputs $\times 16$, $\times 8$, $\times 4$ and $\times 2$ smaller in resolution than the original image, with the final exit as the original reconstruction. Here, the very first early exit outputs a low quality while the final exit outputs higher quality. There has been similar early exit networks used in the literature for various tasks [28, 37].

In `Swift`, we define a loss function at each exit and optimize the training objective of the decoder at all exits. The decoder is trained by introducing additional `L1` reconstruction

---
[5]here the depth refers to the number of layers in the neural network.

losses, so that the outputs of each of the early-exit heads minimizes the difference with the original input (I). The objective function is as shown below:

$$\mathcal{L}_{\text{rec}} = \frac{1}{L}\sum_{i=0}^{L-1}\big[\underbrace{\|\mathcal{D}(c_i) - r_i\|_1}_{\text{residual quality loss}} + \frac{1}{S+1}\sum_{j=0}^{S}\underbrace{\left\|\mathcal{D}_{hd_j}^{\text{ss}}(\oplus_{k=0}^i c_k) - I\right\|_1}_{\text{image quality loss}}\big]$$

Here, S is the number of exits. Given a combination of these multiple exits and downloaded codes, the decoder outputs the quality corresponding to both dimensions. Figure 10 shows the heatmap of average video quality when the client decodes

Figure 10: Quality matrix as a function of exit depth and the number of layered codes.

different number layered codes while exiting at different depths, for UVG videos described in §6.1. For example, if the client only fetches the base layer (shown as 1 in the figure) and exits at depth 1, the quality of the decoded segment is 28dB. However, if the client decodes 4 layers and decodes to completion (exit at depth 4), the quality of the decoded segment increases to 32 dB. Note here that the number of layered codes that can be fetched depends on the network capacity while the exit depth depends on the compute capacity.

At runtime, `Swift` decoder decides on when to exit depending on the GPU capacity. The GPU capacity determines the latency in decoding a segment by computing until different depths. Given a GPU capacity, `Swift` chooses the maximum depth such that the segment will be decoded without incurring any stalls because the buffer is empty. In §5 we discuss how the client chooses the decoder and obtains the relationship between decode latency and exit depth.

4.2 Adapting ABR for Layered Neural Codecs

A traditional ABR algorithm [31, 35, 52] using regular codecs takes as input the available throughput, buffer levels, and details about the future video segments. The algorithm outputs the quality of the video segment to download next. `Swift` needs to adapt existing ABR algorithms to work with layered neural codecs. We describe this adaptation in terms of changes to ABR's output, input, and the objective function. We then discuss how we instantiate the ABR algorithm with these changes. See Figure 11 for an overview. In the discussion below, we assume that the ABR algorithm is run at the server; but it can be adapted to run at the client.

Output: The crucial change to `Swift`'s ABR is that unlike traditional ABR, our algorithm can make one of two choices: download the base layer (i.e., the code with lowest quality) of a future segment or, download an enhancement layer of one of the buffered video segments (that is not played yet) to
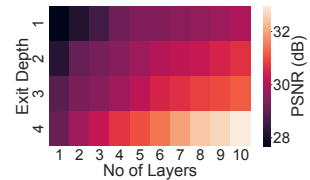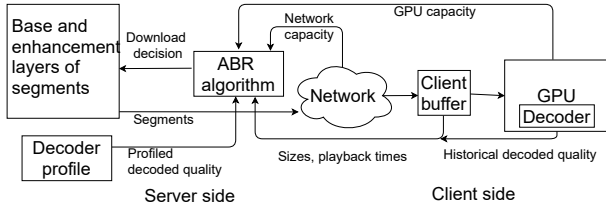
Figure 11: Swift's video streaming pipeline.

enhance quality. It makes this determination based on a (new) set of inputs and the objective function. This change to the output is needed for streaming algorithms that use layered code, including Grad [31], the state-of-the-art streaming that uses layered coding. We compare the performance of Swift to Grad in §6.

Input: Swift introduces two additional inputs to the ABR to take into account the compute variability while decoding. ABR takes as input a matrix that maps the quality of the decoded segment against compute capacity needed for this decoding (similar to Figure 10). This quality matrix is generated offline at the server for each video segment for different compute capacities i.e., for all combinations of layers and exit depths. The next input is the current GPU capacity at the client. Since we run the ABR algorithm at the server, this information is sent by the client in the segment request packet. If the ABR algorithm is run on the client, the GPU capacity is readily available there. In this case the quality matrix at the server can be sent as a part of the manifest file.

The second change is with respect to segments downloaded/buffered at the client but not yet played. Swift's ABR needs information about these segments to makes it choice. Specifically, this includes i) the size of the remaining layers for current/buffered segments, ii) the playback time of all segments in the buffer, to determine if the segment can be enhanced before playback. In addition to that, we input the history of the decoded quality of the last k displayed segments to reduce variation in quality (this is often called smoothness and is an important metric for improving QoE).

Objective function: The video Quality of Experience (QoE) is typically captured using three metrics: i) playback quality (Q), i.e., the quality of the downloaded segments and ii) rebuffering ratio (R) that measures how often the video stalls because the buffer is empty, iii) smoothness (S) that measures fluctuation in quality. Formally, $QoE = Q - \alpha R - \beta S$, where α and β are the coefficients to control the penalties of rebuffering and smoothness [31, 35, 52]. Since Swift's decoding performance is variable, it may not always provide the best quality that is possible from the downloaded segments. So instead, the objective function in Swift takes into account the quality of the *decoded segment* (Q_d) instead of the downloaded segment. Similarly, we compute the smoothness from the quality of decoded segments (S_d) instead of downloaded qualities.

5 Implementation and System Setup

Swift's end-to-end implementation includes its layered neural codecs and the ABR protocol presented in §3.1 and §4.

5.1 Layered Codec Implementation

We implement our layered coding on top of VCII [60]. VCII is a state-of-the-art neural codec that is learnt over an Autoencoder network. VCII achieves compression efficiency close to state-of-the-art regular (non-neural) video codecs. Similar to regular codecs, VCII does not produce layered codes. Instead, we implement the layered encoder over VCII as described in §3.2. For the decoder at the client, similarly, we modify the loss function to incorporate the single-shot decoding and multi-exit decoder capability. Since our layered technique can be applied as a general extension to any codec, we do not need to change the internal codec logic.

After designing the new encoder/decoder over VCII, the encoder and decoder is retrained for 100K iterations on an Nvidia RTX 2070 GPU. We use ADAM optimizer with a batch size of 16. During the training, we use multiple randomly cropped 64×64 image patches from the original images for generalization purposes. The model is trained to produce up to 10 layered codes.

For training, we use the Kinetics dataset [13]. It has 37K videos. We train on 27K, test on 10K videos. For more rigorous testing, we also test on completely different datasets (more details about testing in §6). The training takes around 6 hours. Since the training will be done offline and only once, the training time is reasonable.

5.2 Streaming Implementation

We implement our adapted ABR by modifying Pensieve [35]. For training the ABR model, we use $k = 10$ throughput and compute capacity measurements passing through a 1D-CNN with 128 filters, the quality matrix passed through a 2D-CNN, and aggregated with other inputs described in §4.2. The learning rate and discount factor for the network are 0.001 and 0.99 respectively. We run the ABR algorithm every time a segment or its layer (s) is downloaded. We train the model using simulated network and compute traces, similar to that used in Pensieve [35].

We run the ABR algorithm at the server. Similar to other video streaming servers, the Swift server processes the video segments and encodes them. The server also performs finegrained profiling of the decoder for two bits of information. First, it creates a matrix of quality levels for different depths. Second, it creates a mapping between GPU capacities and time taken to finish decoding until different depths. Both of this are used as input to the adapted ABR algorithm.

6 `Swift` Evaluation

We evaluate `Swift` both in terms of end-to-end streaming and coding performance. We compare `Swift` with a suite of streaming algorithms and its layered coding with commonly used regular codec (HEVC) [68] and layered codec (SHVC) [3]. Our evaluation shows that:

- Overall QoE with `Swift` improves by 45% at the median compared to the second best streaming performance.
- `Swift` uses 16% and 22% less bandwidth compared to next best streaming algorithms that use regular and layered codecs, respectively.
- `Swift`'s neural layered codec improves compression efficiency by 58% over state-of-the-art layered codec SHVC.

6.1 Evaluation Methodology

In this section, we describe the methodology to evaluate `Swift`'s end-to-end streaming performance.

Experimental set up. We conduct all experiments on a desktop with Nvidia 2070 RTX GPU as the client. Our evaluation uses FullHD videos from UVG [5] dataset consisting of 7 videos for streaming.[6] Each video is of 5 mins and is divided into 5 second segments. Each experiment runs for all segments in the video emulated over network capacity and compute capacity traces (described below). The performance is reported as an average across all the segments in the video.

Network and compute conditions. Most of our evaluation is over real traces collected by FCC [8], similar to recent video streaming works [35, 69]. We use 500 traces and filter the traces to have a minimum bandwidth of 1 Mbps. After filtering, FCC dataset has an average bandwidth of 8.2 Mbps with a standard deviation of 3.6 Mbps. These traces capture real world network throughput variations.

Unlike other video streaming approaches, `Swift` is affected by compute capacity. To stress test our system, we evaluate `Swift` by synthetically varying the client's GPU capacity. We modify the GPU capacity by choosing a random number of the processes to be active in each time slot; the number of processes active is modeled as a Poisson distribution with $\lambda = 5$. Each process shares the GPU equally and we constrain the maximum number of processes to 5.

An ideal scenario for `Swift` is when the GPU capacity is fixed and 100% of the GPU is available. For completeness, we run experiments under this condition. We refer to this as `Swift`-C in the graphs. For a fair comparison, we compare `Swift` with existing methods assuming they have hardware accelerated decoding, while varying GPU resources for `Swift`.

Metrics. We measure streaming performance using the following metrics: 1) video QoE (as defined in §4.2) normalized

---

6Note that the compression performance is evaluated using a more diverse and standard set of video sequences (see §6.3.1).

against maximum QoE possible and averaged across all segments for all traces, 2) bandwidth usage, 3) reaction time (as defined in §6.2.3).

Baselines. We compare the performance of `Swift` with multiple state-of-the-art streaming algorithms that use different combinations of video coding and ABR algorithms:

- **Grad [31]:** Grad is the state-of-the-art algorithm using layered coding technique (SHVC [12]) combined with ABR. This is the closest system to `Swift`. Grad employs a hybrid coding mechanism with SHVC to minimize the cross layer compression overhead and uses a reinforcement learning-based ABR adapted from Penseive [35].
- **BOLA [52]:** BOLA and the two alternatives below use regular (not layered) codec H.265. BOLA uses an ABR algorithm that maximizes the quality of the video segment based on the buffer levels at the client. BOLA is commonly used in the industry [2].
- **Pensieve [35]:** Pensieve is also built over H.265 [53] but uses a reinforcement learning-based ABR algorithm.
- **BOLA-FS [51]:** BOLA-FS builds over H.265 [53] and uses buffer levels at the client to choose the next video segment, similar to BOLA. However, different from BOLA, BOLA-FS allows video quality upgrades, where low quality segments in the buffer are replaced with higher quality by re-downloading them, when network conditions improve. The problem is that the previously downloaded segments are not used, resulting in wasted bandwidth.

In all of these cases, when using H.265 [53], we encode each segment into six bitrates:{1Mbps, 5Mbps, 8Mbps, 12Mbps, 16Mbps}. For Grad, which uses scalable coding, we encode the video in six layers to achieve similar quality levels. We note that in both cases, encoding the videos into 6 quality levels provided the best results. In case of H.265, it does not support fine-grained adaptation to work well with more quality levels. In case of Grad/SHVC, the compression overhead is too high when using more quality levels. For `Swift`, we encode up to 10 layers for more flexible adaptation as there is no compression overhead.

6.2 End-to-end Streaming Results

6.2.1 End-to-end QoE Results

Figure 12 shows the overall QoE of `Swift` compared with the four alternatives, along with `Swift`-C. `Swift`-C represents the best possible performance of `Swift`, when compute capacity does not vary and GPU availability is 100%.

We first compare `Swift` with Grad and BOLA-FS which can both upgrade quality of the buffered video segments when network conditions improve. `Swift` improves QoE by 43% and 48% compared to Grad and BOLA-FS respectively. In the case of Grad, the problem is the high compression overhead incurred in implementing layered coding (§2.3). In case of BOLA-FS, there is a significant bandwidth wastage. When
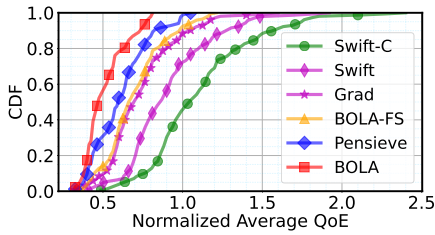
Figure 12: End-to-end QoE. `Swift` improves QoE by 45% at the median compared to the second best performing algorithm.
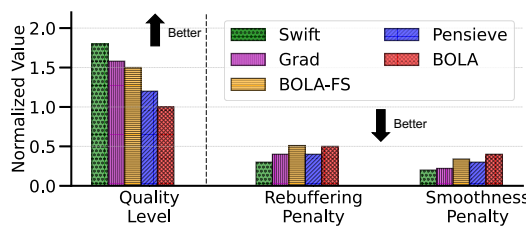
Figure 13: Breakdown of QoE. Overall, `Swift` has higher quality level while having less rebuffering and smoothness penalty.
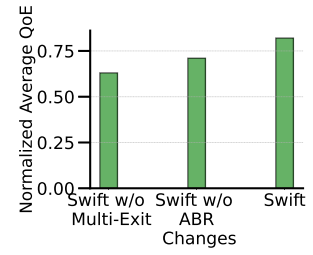
Figure 14: Breakdown `Swift`'s performance with its individual components.
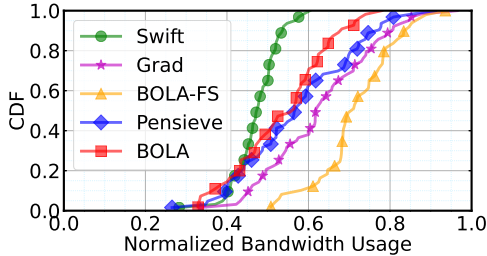
Figure 15: Bandwidth usage of `Swift` compared to state-of-the-art streaming systems. `Swift` improves bandwidth usage especially with respect to systems that upgrade quality when the network conditions improve, namely, BOLA-FS and Grad.

compared to Pensieve and BOLA, `Swift` outperforms by 67% and 74% respectively. Both Pensieve and BOLA do not upgrade video segment quality when the network improves, resulting in poorer quality.

`Swift`-C (Figure 12) shows the QoE achieved when GPU is not fluctuating and 100% of the GPU is used. As expected, `Swift`-C outperforms `Swift` under varying GPU. We also evaluated `Swift` under WiFi (802.11ac) network (client-server RTT: 20ms) without throttling the bandwidth and Nvidia 2070 GPU at 100%. We find that, `Swift` still outperforms the next-best-performing algorithm by 28%.

QoE breakdown Figure 13 shows the performance of the five streaming algorithms in terms of each QoE component: average quality of the video segments, rebuffering, and smoothness penalty. `Swift` improves average quality by 19% compared to the next-best streaming alternative. `Swift` also decreases rebuffering and smoothness penalty by 8% and 11%, respectively, compared to the next best streaming alternative.

Ablation study Figure 14 shows the impact of `Swift`'s components: 1) `Swift` without multi-exit, 2) `Swift` without adapted ABR. The figure shows that both components are critical to the performance of `Swift`. `Swift` without multi-exit performs poorly compared to the full `Swift` because the decoder runs through the entire network even when GPU capacity is low rather than exit early. This results in high decoding latency and in-turn high video stalls. In case of `Swift` without adapting ABR, the system performs poorly because it only adapts to network variations and not compute variations.

6.2.2 Bandwidth Benefits

Figure 15 shows the bandwidth benefits of `Swift` over existing streaming alternatives. `Swift` uses 16% and 22% less bandwidth compared to Grad and BOLA-FS, incurred due to compression overhead and wasted bandwidth respectively. Pensieve and BOLA results in comparatively less bandwidth waste, but cannot upgrade quality when the network improves resulting in poorer video quality (Figure 12).

6.2.3 Reaction to Bandwidth Fluctuations

One key advantage of `Swift`, or layered coding in general, is that it can adapt to bandwidth fluctuation without wasting bandwidth (see Figure 3). To illustrate this, we use an example network trace that starts with an average low bandwidth of 1 Mbps for 100 seconds and increases to average 18 Mbps for the rest of the trace (250 secs).

To compare the performance of these different streaming techniques, we measure the reaction time in two ways: 1) reaction time to any quality (RTA), which is the elapsed time between when the bandwidth increases to when the user experiences any higher quality video, 2) reaction time to highest quality (RTH), which is the elapsed time between when the bandwidth increases to when the user experiences the highest sustainable video quality.

Figure 16 shows one scenario how the different streaming algorithms adapt to changing network condition for a 250 second sample trace. The black line shows the change in throughput. `Swift` is the first to react to the change in throughput of all the other alternatives. Figure 17 shows qualitatively that `Swift` reacts faster, both in terms of RTA and RTH, compared to the alternatives.

Overall, the normalized average video segment quality of `Swift` throughout the trace was 1.8 compared to the next best alternative, which was 1.6. The reaction time is low even when the throughput decreases instead of increasing (not shown).

6.3 Compression Results

We compare `Swift`'s codec with:

- **HEVC [53]:** This is the most commonly used video codec for video streaming today. We use the `libx265` library
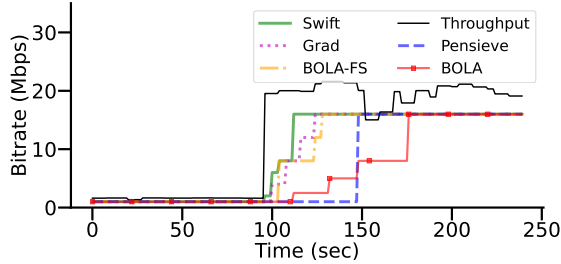
Figure 16: `Swift` reacts faster compared to all other alternatives. Throughput changes at the 100 second mark.

(a) Reaction time to any quality   (b) Reaction time to highest quality
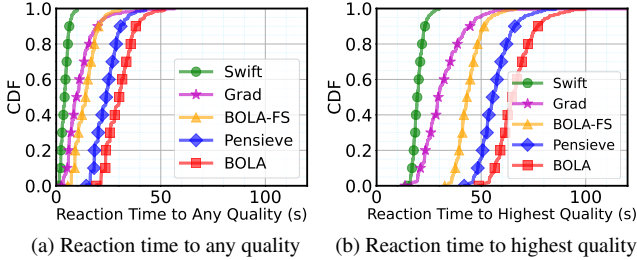
Figure 17: Reaction time. `Swift` reacts 2× and 2.5× faster than Grad and BOLA-FS to reach highest quality, and 25% and 40% faster to reach any high quality.

from FFMPEG [7]. We did not investigate the latest coding standard VVC [59] as it is still in its early stage. We report H.265 results with commonly used codec configuration.[7]

- **Scalable HEVC (SHVC) [12]:** This is a state-of-the-art layered coding method, built as a scalable extension of HEVC also known as SHVC [12]. We evaluate SHVC using a reference implementation from [1].

We present the result averaged over three datasets. One dataset is the test set from the Kinetics dataset (§5). The other two datasets are VTL [6] and UVG [5] that are not used in training. All VTL test videos are in 352×288 and UVG videos are in 1920×1080 resolution.

6.3.1 Compression Efficiency

Figure 18 shows the video quality vs. video size in terms of bits per pixel (BPP) after compression. The metrics we use are: 1) PSNR – this computes the peak signal to noise ratio between two images (higher PSNR indicates better quality of reconstruction), and 2) MS-SSIM (multi-scale structural similarity index method) – a perceptual quality metric taking into account the structural information to weigh more on the spatially close pixels with strong inter-dependencies [56]. For SHVC and `Swift`'s layered coding a total of 6 layers are used to produce the plots – each point refers to the joint performance of all 6 layers. For HEVC or H.265, each quality point is encoded independently with a different bitrate.

`Swift`'s layered coding achieves 58% better compression on average compared to traditional layered coding (SHVC).

---

[7]We use *fast* preset with group of pictures value 30.

(a) PSNR   (b) MS-SSIM
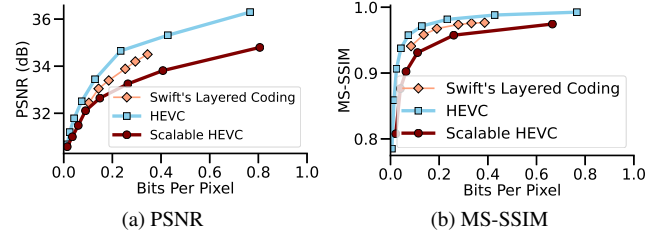
Figure 18: Compression efficiency. `Swift`'s Layered coding outperforms the traditional layered coding SHVC and performs close to HEVC.

(a) Encoding Latency   (b) Decoding Latency
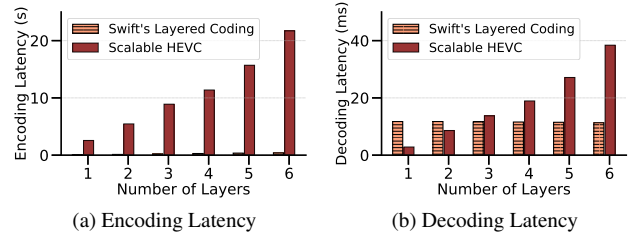
Figure 19: Encoding and Decoding latency. Both the encoding and decoding latency `Swift`'s layered coding is significantly less than traditional layered coding. More importantly, the decoding latency of `Swift`'s layered coding is independent of number of layered codes unlike traditional layered coding.

The compression difference is mainly due to the cross-layer overhead incurred by SHVC. `Swift`'s compression is close to that of HEVC with `Swift` performing poorer by 0.02 MS-SSIM and 0.8 dB in the median case. However, the QoE when using `Swift` is still better than the QoE than HEVC because of the fine-grained rate adaptation benefits.

6.3.2 Encoding and Decoding Latency

In this set of experiments, we compare the encoding and decoding latency of `Swift`'s codec and the state-of-the-art layered codec (we omit regular HEVC codec here because it has negligible latencies). The evaluation here benchmarks the latency on a Desktop machine (Intel 12 core CPU with Nvidia RTX 2070 GPU).[8] Figure 19 shows average per-frame encoding and decoding latency as a function of number of layers. There is 15× increase in encoding latency from layer one to six in case of traditional layered coding (SHVC). `Swift`'s layered coding has an encoding latency of 20ms for one layer and increases proportionately by 6× for the sixth layer. While improvements are still needed to get close to the encoding latency of regular (non-layered) codec, the encoding latency in `Swift` is significantly less than SHVC. More importantly, the decoding latency of `Swift`'s layered coding is independent of the number of layers, while SHVC increases proportionately similar to encoding latency. This is due to the use of the single-shot decoder of `Swift` (§3.3).

---

[8]Of note, HEVC and SHVC are run on the CPU and our layered coding runs on GPU as it is the most efficient on the GPU.

7 Discussion

`Swift`'s layered coding addresses compression overhead challenge highlighted in §2. Below we discuss some of the additional benefits as well as limitations of layered neural codecs.

7.1 Additional Opportunities of Neural Codecs

Flexible data-driven approach: In a learning-based approach to video coding the learning can be made video specific, for example, customized to video types [9], likely providing an opportunity to better learn video type-specific features. This ultimately leads to streaming quality improvements relative to the one-size-fits-all solution that exists today.

Software-defined coding: Unlike existing codecs, neural codecs do not need to be baked into a fixed hardware. They are more easily upgradable. Common ML tools (e.g., Pytorch with CUDA support) ensure that running neural codecs on data parallel co-processors (e.g., GPUs) requires significantly less development cycle compared to porting traditional codecs. The softwarization of video coding gives the content providers flexibility to integrate codec features on demand, support agile codec development, provide royalty-free codecs, and eliminate compatibility issues.

Design of application-specific codecs: Various video analytics solutions [21, 24, 62] often apply DNN-based analytics (e.g., object detection and classification) on video streams that are coded using traditional video codecs. However, this results in suboptimal performance because they are originally designed for human perceptual quality. Instead, neural codecs are amenable to training with loss functions more tuned towards appropriate analytics. Similarly, specialized codecs can be designed for conferencing or surveillance that may have constant backgrounds or other commonly appearing features that, once learnt, can be compressed very efficiently.

7.2 Limitations of Neural Codecs

One key assumption of `Swift` is that the client devices need to be equipped with GPUs or other similar accelerators to run neural networks. Otherwise, the decoding latency could become a bottleneck. While such accelerators are expected to be commonplace, they do add to the device cost and energy consumption. Also, the current design of `Swift` targets on-demand video streaming because the iterative layered coding does not offer real-time encoding. More work is needed on the encoding side for applying `Swift` to live video applications (such as conferencing or live analytics) to overcome the encoding latency challenges.

Finally, the QoE evaluations for `Swift` are done using a learning based ABR algorithm based on Pensieve [35] and Grad [31]. It may be challenging to generalize such algorithms for unknown environments that can still occur in practice [34, 63]. However, given our characterization of the input and output along with the objective function, we expect that other algorithmic ABR approaches (such as BOLA [52] or FUGU [63]) are equally applicable for `Swift`.

8 Related Work

Video streaming: There has been an extensive prior work on improving QoE for regular video streaming. Much of the previous work focuses on improving the adaptive bitrate algorithms by better predicting the available throughput. Festive [25] predicts throughput using a harmonic mean. BBA [23] and BOLA [52] take into account the buffer capacity to determine video bitrate. Fugu [63] and MPC [66] use learning-based throughput prediction. There is a recent interest in using reinforcement learning for adaptive bitrate selection (e.g., Pensieve [35] and other follow-up work). Recent solutions such as SENSEI [69] improves QoE by introducing user sensitivity into ABR algorithms. `Swift` is able to extend existing ABR algorithms for use with neural codecs and can synergistically optimize network and compute resources to improve QoE.

Video compression: Traditional compression methods such as H.264/265 [53, 57] employ many algorithms that include frame prediction [64, 70], transform coding and quantization [18, 40, 48, 58], and entropy coding [54]. In the past decade or two, there have been several studies on improving both the compression efficiency and coding latency for these algorithms on an individual basis [15, 16, 26, 49]. Similarly, there have been extensive studies on improving the traditional layered coding, while still facing challenges of compression overhead and high latency [11, 14, 67]. Unlike all these algorithmic codecs, there is a recent shift in codec design using deep learning [32, 43, 44, 60]. `Swift` belongs to this second category and develops layered coding on top of neural codecs.

9 Conclusions

We have described `Swift`, an adaptive video streaming system using layered neural codecs that use deep learning. `Swift`'s neural codec achieves efficient layered compression without introducing cross layer compression overheads and eliminates the dependency of decoding latency on the number of layers. `Swift` extends existing ABR frameworks to accommodate layered neural codecs and demonstrates significant performance benefits compared to state-of-the-art adaptive video streaming systems.

Acknowledgements

References

[1] A Reference Implementation of SHVC (Scalable extentino to HEVC). https://hevc.hhi.fraunhofer.de/shvc.

[2] Akamai players. https://players.akamai.com/players/dashjs.

[3] HEVC scalability extension. https://hevc.hhi.fraunhofer.de/shvc.

[4] libvpx-vp9. https://trac.ffmpeg.org/wiki/Encode/VP9.

[5] Ultra video group. http://ultravideo.fi/.

[6] Video trace library. http://trace.eas.asu.edu/index.html.

[7] x265. https://trac.ffmpeg.org/wiki/Encode/H.265.

[8] Measuring broadband America, FCC. https://www.fcc.gov/reports-research/reports/measuring-broadband-america/raw-data-measuring-broadband-america-eighth, 2018.

[9] Sami Abu-El-Haija, Nisarg Kothari, Joonseok Lee, Paul Natsev, George Toderici, Balakrishnan Varadarajan, and Sudheendra Vijayanarasimhan. Youtube-8m: A large-scale video classification benchmark. *preprint arXiv:1609.08675*, 2016.

[10] Shivang Aggarwal, Urjit Satish Sardesai, Viral Sinha, Deen Dayal Mohan, Moinak Ghoshal, and Dimitrios Koutsonikolas. LiBRA: learning-based link adaptation leveraging PHY layer information in 60 GHz WLANs. In *ACM Confernece on Emerging Networking Experiments and Technologies*, pages 245–260, 2020.

[11] Ghufran Baig, Jian He, Mubashir Adnan Qureshi, Lili Qiu, Guohai Chen, Peng Chen, and Yinliang Hu. Jigsaw: Robust live 4K video streaming. In *MobiCom*, pages 1–16, 2019.

[12] Jill M Boyce, Yan Ye, Jianle Chen, and Adarsh K Ramasubramonian. Overview of shvc: Scalable extensions of the high efficiency video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 26(1):20–34, 2015.

[13] Joao Carreira and Andrew Zisserman. Quo vadis, Action recognition? A new model and the kinetics dataset. In *CVPR*. IEEE, 2017.

[14] Jacob Chakareski, Sangeun Han, and Bernd Girod. Layered coding vs. multiple descriptions for video streaming over multiple paths. *Multimedia Systems*, 10(4):275–285, 2005.

[15] Mei-Juan Chen, Yu-De Wu, Chia-Hung Yeh, Kao-Min Lin, and Shinfeng D Lin. Efficient CU and PU decision based on motion information for interprediction of HEVC. *IEEE Transactions on Industrial Informatics*, 14(11):4735–4745, 2018.

[16] Santiago De-Luxán-Hernández, Valeri George, Jackie Ma, Tung Nguyen, Heiko Schwarz, Detlev Marpe, and Thomas Wiegand. An intra subpartition coding mode for vvc. In *2019 IEEE International Conference on Image Processing (ICIP)*, pages 1203–1207. IEEE, 2019.

[17] Anis Elgabli, Vaneet Aggarwal, Shuai Hao, Feng Qian, and Subhabrata Sen. LBP: robust rate adaptation algorithm for SVC video streaming. *IEEE/ACM Transactions on Networking*, 26(4):1633–1645, 2018.

[18] Vivek K Goyal. Theoretical foundations of transform coding. *IEEE Signal Processing Magazine*, 18(5):9–21, 2001.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[20] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006.

[21] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodik, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B Gibbons, and Onur Mutlu. Focus: Querying large video datasets with low latency and low cost. In *Usenix OSDI*, pages 269–286, 2018.

[22] Hanzhang Hu, Debadeepta Dey, Martial Hebert, and J Andrew Bagnell. Learning anytime predictions in neural networks via adaptive loss balancing. In *AAAI*, volume 33, pages 3812–3821, 2019.

[23] Te-Yuan Huang, Ramesh Johari, Nick McKeown, Matthew Trunnell, and Mark Watson. A buffer-based approach to rate adaptation: Evidence from a large video streaming service. *ACM SIGCOMM Computer Communication Review*, 44(4):187–198, 2015.

[24] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. Chameleon: scalable adaptation of video analytics. In *ACM SIGCOMM*, pages 253–266, 2018.

[25] Junchen Jiang, Vyas Sekar, and Hui Zhang. Improving fairness, efficiency, and stability in http-based adaptive video streaming with festive. *IEEE/ACM Transactions on Networking (ToN)*, 22(1):326–340, 2014.

[26] Y-H Kim, J-W Yoo, S-W Lee, J Shin, J Paik, and H-K Jung. Adaptive mode decision for H.264 encoder. *Electronics letters*, 40(19):1172–1173, 2004.

[27] PoLin Lai, Shan Liu, and Shawmin Lei. Low latency directional filtering for inter-layer prediction in scalable video coding using hevc. In *2013 Picture Coding Symposium (PCS)*, pages 269–272. IEEE, 2013.

[28] Stefanos Laskaridis, Alexandros Kouris, and Nicholas D Lane. Adaptive inference through early-exit networks: Design, challenges and directions. *arXiv preprint arXiv:2106.05022*, 2021.

[29] Athanasios Leontaris and Pamela C Cosman. Drift-resistant snr scalable video coding. *IEEE transactions on image processing*, 15(8):2191–2197, 2006.

[30] Weiyao Lin, Krit Panusopone, David M Baylon, and Ming-Ting Sun. A computation control motion estimation method for complexity-scalable video coding. *IEEE transactions on circuits and systems for video technology*, 20(11):1533–1543, 2010.

[31] Yunzhuo Liu, Bo Jiang, Tian Guo, Ramesh K Sitaraman, Don Towsley, and Xinbing Wang. Grad: Learning for overhead-aware adaptive video streaming with scalable video coding. In *Proceedings of the 28th ACM International Conference on Multimedia*, pages 349–357, 2020.

[32] Guo Lu, Wanli Ouyang, Dong Xu, Xiaoyun Zhang, Chunlei Cai, and Zhiyong Gao. DVC: An end-to-end deep video compression framework. In *CVPR*, pages 11006–11015, 2019.

[33] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient GPU cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 289–304, 2020.

[34] Hongzi Mao, Shannon Chen, Drew Dimmery, Shaun Singh, Drew Blaisdell, Yuandong Tian, Mohammad Alizadeh, and Eytan Bakshy. Real-world video adaptation with reinforcement learning. *arXiv preprint arXiv:2008.12858*, 2020.

[35] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with pensieve. In *ACM SIGCOMM*, pages 197–210. ACM, 2017.

[36] Steven McCanne, Martin Vetterli, and Van Jacobson. Low-complexity video coding for receiver-driven layered multicast. *IEEE journal on selected areas in communications*, 15(6):983–1001, 1997.

[37] Alessandro Montanari, Manuja Sharma, Dainius Jenkus, Mohammed Alloulah, Lorena Qendro, and Fahim Kawsar. ePerceptive: Energy Reactive Embedded Intelligence for Batteryless Sensors. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, pages 382–394, 2020.

[38] Arvind Narayanan, Eman Ramadan, Jason Carpenter, Qingxu Liu, Yu Liu, Feng Qian, and Zhi-Li Zhang. A first look at commercial 5g performance on smartphones. In *Proceedings of The Web Conference 2020*, pages 894–905, 2020.

[39] Arvind Narayanan, Eman Ramadan, Rishabh Mehta, Xinyue Hu, Qingxu Liu, Rostand AK Fezeu, Udhaya Kumar Dayalan, Saurabh Verma, Peiqi Ji, Tao Li, et al. Lumos5g: Mapping and predicting commercial mmwave 5g throughput. In *Proceedings of the ACM Internet Measurement Conference*, pages 176–193, 2020.

[40] Tung Nguyen, Philipp Helle, Martin Winken, Benjamin Bross, Detlev Marpe, Heiko Schwarz, and Thomas Wiegand. Transform coding techniques in hevc. *IEEE Journal of Selected Topics in Signal Processing*, 7(6):978–989, 2013.

[41] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*, 2015.

[42] Amy R Reibman, Leon Bottou, and Andrea Basso. Scalable video coding with managed drift. *IEEE transactions on circuits and systems for video technology*, 13(2):131–140, 2003.

[43] Oren Rippel, Alexander G Anderson, Kedar Tatwawadi, Sanjay Nair, Craig Lytle, and Lubomir Bourdev. Elf-vc: Efficient learned flexible-rate video coding. *arXiv preprint arXiv:2104.14335*, 2021.

[44] Oren Rippel, Sanjay Nair, Carissa Lew, Steve Branson, Alexander G Anderson, and Lubomir Bourdev. Learned video compression. *preprint arXiv:1811.06981*, 2018.

[45] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. Infaas: Automated model-less inference serving. In *2021 Usenix ATC 21*, pages 397–411, 2021.

[46] Swetank Kumar Saha, Shivang Aggarwal, Rohan Pathak, Dimitrios Koutsonikolas, and Joerg Widmer. MuSher: An Agile Multipath-TCP Scheduler for Dual-Band 802.11 ad/ac Wireless LANs. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–16, 2019.

[47] Heiko Schwarz, Detlev Marpe, and Thomas Wiegand. Overview of the scalable video coding extension of the H.264/AVC standard. *IEEE Transactions on circuits and systems for video technology*, 17(9):1103–1120, 2007.

[48] Heiko Schwarz, Tung Nguyen, Detlev Marpe, and Thomas Wiegand. Hybrid video coding with trellis-coded quantization. In *2019 Data Compression Conference (DCC)*, pages 182–191. IEEE, 2019.

[49] Mahmut E Sinangil, Vivienne Sze, Minhua Zhou, and Anantha P Chandrakasan. Cost and coding efficient

motion estimation design considerations for high efficiency video coding (HEVC) standard. *IEEE Journal of selected topics in signal processing*, 7(6):1017–1028, 2013.

[50] Iraj Sodagar. The MPEG-DASH standard for multimedia streaming over the internet. *IEEE MultiMedia*, (4), 2011.

[51] Kevin Spiteri, Ramesh Sitaraman, and Daniel Sparacio. From theory to practice: Improving bitrate adaptation in the dash reference player. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 15(2s):1–29, 2019.

[52] Kevin Spiteri, Rahul Urgaonkar, and Ramesh K Sitaraman. Bola: Near-optimal bitrate adaptation for online videos. *IEEE/ACM Transactions on Networking*, 28(4):1698–1711, 2020.

[53] Gary J Sullivan, Jens-Rainer Ohm, Woo-Jin Han, and Thomas Wiegand. Overview of the high efficiency video coding (HEVC) standard. *IEEE Transactions on circuits and systems for video technology*, 22(12):1649–1668, 2012.

[54] Vivienne Sze and Madhukar Budagavi. High Throughput CABAC Entropy Coding in HEVC. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12):1778–1791, 2012.

[55] George Toderici, Damien Vincent, Nick Johnston, Sung Jin Hwang, David Minnen, Joel Shor, and Michele Covell. Full resolution image compression with recurrent neural networks. In *CVPR*, pages 5306–5314, 2017.

[56] Zhou Wang, Eero P Simoncelli, and Alan C Bovik. Multiscale structural similarity for image quality assessment. In *The Thrity-Seventh Asilomar Conference on Signals, Systems & Computers, 2003*, volume 2, pages 1398–1402. Ieee, 2003.

[57] Thomas Wiegand, Gary J Sullivan, Gisle Bjontegaard, and Ajay Luthra. Overview of the H.264/AVC video coding standard. *IEEE Transactions on circuits and systems for video technology*, 13(7):560–576, 2003.

[58] Mathias Wien. Variable block-size transforms for H.264/AVC. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):604–613, 2003.

[59] Mathias Wien and Benjamin Bross. Versatile video coding–algorithms and specification. In *2020 IEEE International Conference on Visual Communications and Image Processing (VCIP)*, pages 1–3. IEEE, 2020.

[60] Chao-Yuan Wu, Nayan Singhal, and Philipp Krahenbuhl. Video compression through image interpolation. In *ECCV*, pages 416–431, 2018.

[61] Feng Wu, Shipeng Li, and Ya-Qin Zhang. A framework for efficient progressive fine granularity scalable video coding. *IEEE transactions on Circuits and Systems for Video Technology*, 11(3):332–344, 2001.

[62] Xiufeng Xie and Kyu-Han Kim. Source compression with bounded dnn perception loss for iot edge computer vision. In *ACM MobiCom*, pages 1–16, 2019.

[63] Francis Y Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and Keith Winstein. Learning in situ: a randomized experiment in video streaming. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 495–511, 2020.

[64] Jiheng Yang, Baocai Yin, Yanfeng Sun, and Nan Zhang. A block-matching based intra frame prediction for H.264/AVC. In *2006 IEEE International Conference on Multimedia and Expo*, pages 705–708. IEEE, 2006.

[65] Ren Yang, Fabian Mentzer, Luc Van Gool, and Radu Timofte. Learning for video compression with hierarchical quality and recurrent enhancement. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6628–6637, 2020.

[66] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. A Control-theoretic Approach for Dynamic Adaptive Video Streaming over HTTP. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 325–338. ACM, 2015.

[67] Sangki Yun, Daehyeok Kim, Xiaofan Lu, and Lili Qiu. Optimized layered integrated video encoding. In *INFOCOM*, pages 19–27. IEEE, 2015.

[68] Alireza Zare, Alireza Aminlou, Miska M Hannuksela, and Moncef Gabbouj. HEVC-compliant tile-based streaming of panoramic video for virtual reality applications. In *Proceedings of the 24th ACM international conference on Multimedia*, pages 601–605. ACM, 2016.

[69] Xu Zhang, Yiyang Ou, Siddhartha Sen, and Junchen Jiang. Sensei: Aligning video streaming quality with dynamic user sensitivity. In *NSDI*, pages 303–320, 2021.

[70] Shiping Zhu, Shupei Zhang, and Chenhao Ran. An improved inter-frame prediction algorithm for video coding based on fractal and H.264. *IEEE Access*, 5:18715–18724, 2017.
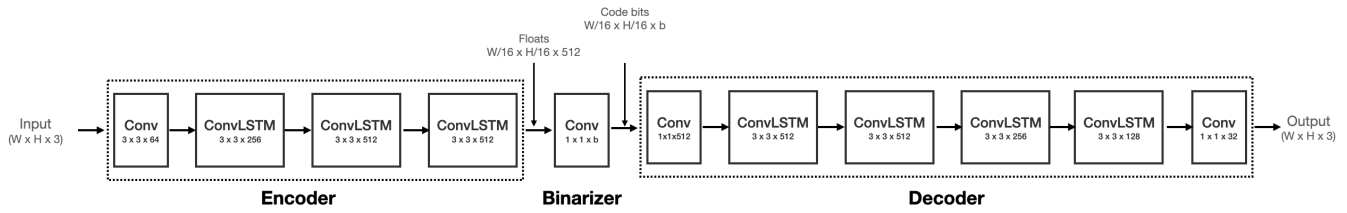
Figure 20: An example of a base neural codec and its internal logic to encode and decode a video frame in a single iteration.

A   Appendix. Example Neural Codec

`Swift`'s layered neural coding is designed as a generic extension that can be implemented on top of any neural codec, and hence throughout the paper, we considered neural codec as a blackbox without discussing the internal details of codec logic. In this section, we present an example codec for a better understanding of neural coding principles. This example follows the design of VCII [60], the same design we also used in our implementation.

In general, most of the existing neural codecs follow traditional concepts of I, P, and B frames when compressing a video [22, 43, 60]. An I frame is compressed much like an image with no reference, and P/B frames reference other frames for reconstruction as they encode motion and residual information relative to the reference frames. `Swift` adopts a similar approach of compressing I frames and P/B frames separately by using i) a neural image codec [55] for compressing I frames, and ii) a neural video codec [60] for compressing P/B frames. The output for each of these frames after the encoding stage from the Autoencoder, is a neural representation,

i.e., code bits with floating point values. The code bits for I frames represent directly the frame data, however, the code bits P/B frames represent motion and residual information with respect to the reference frames.

Figure 20 shows an example codec structure followed by [55, 60] as well as `Swift`. It contains three key parts: encoder, binarizer, and decoder. The encoder takes the original video frame as input and applies convolutions (along with an LSTM block) to downscale the frame into a low dimensional vector. In our example, we have four such blocks, each downscaling the frame resolution by half. For example, when we encode a 1280×720 frame, the output of encoding stage contains $80 \times 45 \times 512$ resolution with floating point representations. After the encoding, a binarizer converts the floats to a binary bitstream with the same resolution but packs each float in b bits. Optionally, these bits can be further passed through an entropy encoder [54] to compress the bitstream efficiently. During the decoding process, a reverse process is learned by upsampling the frame resolution at each stage in the network, achieving the original resolution at the final stage.

Ekya: Continuous Learning of Video Analytics Models on Edge Compute Servers

Romil Bhardwaj[1,2], Zhengxu Xia[3], Ganesh Ananthanarayanan[1], Junchen Jiang[3], Yuanchao Shu[1], Nikolaos Karianakis[1], Kevin Hsieh[1], Paramvir Bahl[1], and Ion Stoica[2]

[1]Microsoft, [2]UC Berkeley, [3]University of Chicago

Abstract

Video analytics applications use edge compute servers for processing videos. Compressed models that are deployed on the edge servers for inference suffer from *data drift* where the live video data diverges from the training data. Continuous learning handles data drift by periodically retraining the models on new data. Our work addresses the challenge of jointly supporting inference and retraining tasks on edge servers, which requires navigating the fundamental tradeoff between the retrained model's accuracy and the inference accuracy. Our solution Ekya balances this tradeoff across multiple models and uses a micro-profiler to identify the models most in need of retraining. Ekya's accuracy gain compared to a baseline scheduler is 29% higher, and the baseline requires 4× more GPU resources to achieve the same accuracy as Ekya.

1 Introduction

Video analytics applications, such as for urban mobility [2, 5] and smart cars [27], are being powered by deep neural network (DNN) models for object detection and classification, e.g., Yolo [36], ResNet [39] and EfficientNet [61]. Video analytics deployments stream the videos to *edge servers* [14, 15] placed on-premise [13, 38, 81, 84]. Edge computation is preferred for video analytics as it does not require expensive network links to stream videos to the cloud [81], while also ensuring privacy of the videos (e.g., many European cities mandate against streaming their videos to the cloud [11, 87]).

Edge compute is provisioned with limited resources (e.g., with weak GPUs [14, 15]). This limitation is worsened by the mismatch between the growth rate of the compute demands of models and the compute cycles of processors [12, 90]. As a result, edge deployments rely on *model compression* [67, 86, 94]. The compressed DNNs are initially trained on representative data from each video stream, but while in the field, they are affected by *data drift*, i.e., the live video data diverges significantly from the data that was used for training [23, 52, 77, 79]. Cameras in streets and smart cars encounter varying scenes over time, e.g., lighting, crowd densities, and changing object mixes. It is difficult to exhaustively cover all

these variations in the training, especially since even subtle variations affect the accuracy. As a result, there is a sizable drop in the accuracy of edge DNNs due to data drift (by 22%; §2.3). In fact, the fewer weights and shallower architectures of compressed DNNs often make them unsuited to provide high accuracy when trained with large variations in the data.

Continuous model retraining. A promising approach to address data drift is continuous learning. The edge DNNs are incrementally *retrained* on new video samples even as some earlier knowledge is retained [28, 83]. Continuous learning techniques retrain the DNNs periodically [72, 93]; we refer to the period between two retrainings as the "retraining window" and use a sample of the data that is accumulated during each window for retraining. Such ongoing learning [42, 89, 96] helps the compressed models maintain high accuracy.

Edge servers use their GPUs [15] for DNN inference on many live video streams (e.g., traffic cameras in a city). Adding continuous training to edge servers presents a tradeoff between the live inference accuracy and drop in accuracy due to data drift. Allocating more resources to the retraining job allows it to finish faster and provide a more accurate model sooner. At the same time, during the retraining, taking away resources from the inference job lowers its accuracy (because it may have to sample the frames of the video to be analyzed).

Central to the resource demand and accuracy of the jobs are their *configurations*. For retraining jobs, configurations refer to the hyperparameters, e.g., number of training epochs, that substantially impact the resource demand and accuracies (§3.1). The improvement in accuracy due to retraining also depends on *how much* the characteristics of the live videos have changed. For inference jobs, configurations like frame sampling and resolution impact the accuracy and resources needed to keep up with analyzing the live video [22, 37].

Problem statement. We make the following decisions for retraining. (1) in each retraining window, decide which of the edge models to retrain; (2) allocate the edge server's GPU resources among the retraining and inference jobs, and (3) select the configurations of the retraining and inference jobs. We also constraint our decisions such that the inference ac-

curacy *at any point in time* does not drop below a minimum value (so that the outputs continue to remain useful to the application). Our objective in making the above three decisions is to maximize the inference accuracy *averaged over the retraining window* (aggregating the accuracies during and after the retrainings). Maximizing inference accuracy over the retraining window creates new challenges as it is different from (*i*) video inference systems that optimize only the instantaneous accuracy [22, 32, 37], (*ii*) model training systems that optimize only the eventual accuracy [8, 17, 69, 85, 88, 95].

Addressing the fundamental tradeoff between the retrained model's accuracy and the inference accuracy is computationally complex. First, the decision space is multi-dimensional consisting of a diverse set of retraining and inference configurations, and choices of resource allocations over time. Second, it is difficult to know the performance of different configurations (in resource usage and accuracy) as it requires actually retraining using different configurations. Data drift exacerbates these challenges because a decision that works well in a retraining window may not do so in the future.

Solution components. Our solution Ekya has two main components: a resource scheduler and a performance estimator.

In each retraining window, the **resource scheduler** makes the three decisions listed above in our problem statement. In its decisions, Ekya's scheduler prioritizes retraining the models of those video streams whose characteristics have changed the most because these models have been most affected by data drift. The scheduler decides against retraining the models which do not improve our target metric. To prune the large decision space, the scheduler uses the following techniques. First, it simplifies the spatial complexity by considering GPU allocations only in coarse fractions (e.g., 10%) that are accurate enough for the scheduling decisions, while also being mindful of the granularity achievable in modern GPUs [4]. Second, it does not change allocations to jobs *during the retraining*, thus largely sidestepping the temporal complexity. Finally, our micro-profiler (described below) prunes the list of configurations to only the promising options.

To make efficient choices of configurations, the resource scheduler relies on estimates of accuracy after the retraining and the resource demands. We have designed a **micro-profiler** that observes the accuracy of the retraining configurations on a *small subset* of the training data in the retraining window with *just a few epochs*. It uses these observations to extrapolate the accuracies when retrained on a larger dataset for many more epochs. Further, we restrict the micro-profiling to only a small set of *promising* retraining configurations. These techniques result in Ekya's micro-profiler being $100\times$ more efficient than exhaustive profiling while still estimating accuracies with an error of 5.8%. To estimate the resource demands, the micro-profiler measures the retraining duration *per epoch* when 100% of the GPU is allocated, and scales for different allocations, epochs, and training data sizes.

Implementation and Evaluation. We have evaluated Ekya
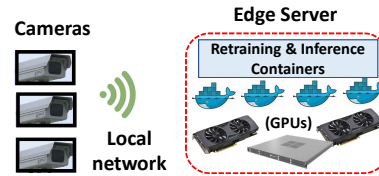
Figure 1: Cameras connect to the edge server, with consumer-grade GPUs for DNN inference and retraining containers.

using a system implementation and trace-driven simulation. We used video workloads from dashboard cameras of smart cars (Waymo [68] and Cityscapes [57]) as well as from traffic and building cameras over 24 hours. Ekya's accuracy compared to competing baselines is 29% higher. As a measure of Ekya's efficiency, attaining the same accuracy as Ekya will require $4\times$ more GPU resources on the edge for the baseline.

Contributions: Our work makes the following contributions.

1) We introduce the metric of *inference accuracy averaged over the retraining window* for continuous training systems.

2) We design an *efficient micro-profiler to estimate* the benefits and costs of retraining edge DNN models.

3) We design a scalable resource scheduler for *joint retraining and inference* on edge servers.

4) We release Ekya's source code and video datasets with 135 hours of videos and corresponding labels to spur future research in continuous learning at the edge. See aka.ms/ekya.

2 Continuous training on edge compute

2.1 Edge Computing for Video Analytics

Video analytics deployments commonly analyze videos on edge servers placed on-premise (e.g., from AWS [14] or Azure [15]). A typical edge server supports tens of video streams [19], e.g., on the cameras in a building, with customized models for each stream [59] (see Figure 1).Video analytics applications adopt edge computing for the following reasons [13, 38, 81].

1) Edge deployments are often in locations where the *uplink network to the cloud is expensive* for shipping continuous video streams, e.g., in oil rigs with expensive satellite network or smart cars with data-limited cellular network. [1]

2) Network links out of the edge locations experience *outages* [76, 81]. Edge compute provides robustness against disconnection to the cloud [26] and prevents disruptions [20].

3) Videos often contain *sensitive and private data* that users do not want sent to the cloud (e.g., many EU cities legally mandate that traffic videos be processed on-premise [11, 87]).

Thus, due to reasons of network cost and video privacy, it is preferred to run both inference and retraining on the edge compute device itself without relying on the cloud. In fact, with bandwidths typical in edge deployments, cloud-based solutions are slower and result in lower accuracies (§6.4).

---

[1]The uplinks of LTE cellular or satellite links is $3-10$Mb/s [58, 65], which can only support a couple of 1080p 30 fps HD video streams whereas a typical deployment has many more cameras [81].

2.2 Compressed DNN Models and Data drift

Advances in computer vision research have led to high-accuracy DNN models that achieve high accuracy with a large number of weights, deep architectures, and copious training data. While highly accurate, using these heavy and general DNNs for video analytics is both expensive and slow [22, 34], which make them unfit for resource-constrained edge computing. The most common approach to addressing the resource constraints on the edge is to train and deploy *specialized and compressed* DNNs [53, 60, 64, 67, 86, 94], which consist of far fewer weights and shallower architectures. For instance, Microsoft's edge video analytics platform [5] uses a compressed DNN (TinyYOLO [75]) for efficiency. Similarly, Google released Learn2Compress[2] for edge devices to automate the generation of compressed models from proprietary models. These compressed DNNs are trained to only recognize the limited objects and scenes specific to each video stream. In other words, to maintain high accuracy, they forego generality for improved compute efficiency [22, 34, 72].

Data drift. As specialized edge DNNs have shallower architectures than general DNNs, they can only memorize limited amount of object appearances, object classes, and scenes. As a result, specialized edge DNNs are particularly vulnerable to *data drift* [23, 52, 77, 79], where live video data diverges significantly from the initial training data. For example, variations in the object pose, scene density (e.g. rush hours), and lighting (e.g., sunny vs. rainy days) over time make it difficult for traffic cameras to accurately identify the objects of interest (cars, bicycles, road signs). Cameras in modern cars observe vastly varying scenes (e.g., building types, crowd sizes) as they move through different neighborhoods and cities. Further, the *distribution* of the objects change over time, which reduces the edge model's accuracy [93, 99]. Due to their ability to memorize limited amount of object variations, edge DNNs have to be continuously updated with recent data and changing object distributions to maintain a high accuracy.

Continuous training. The preferred approach, that has gained significant attention, is for edge DNNs to *continuously learn* as they incrementally observe new samples over time [42, 89, 96]. The high temporal locality of videos allows the edge DNNs to focus their learning on the most recent object appearances and object classes [72, 82]. In Ekya, we use a modified version of iCaRL[89] learning algorithm to on-board new classes, as well as adapt to the changing characteristics of the existing classes. Since manual labeling is not feasible for continuous training systems on the edge, the labels for the retraining are obtained from a "golden model" - a highly accurate (87% and 84% accuracy on Cityscapes and Waymo datasets, respectively) but expensive model (deeper architecture with large number of weights). The golden model cannot keep up with inference on the live videos and we use it to label only a small fraction of the videos in the retraining window. Our approach is essentially that of supervising a

(a) Class Distribution **(b)** Accuracy

(c) Accuracy vs data drift **(d)** Person class variations
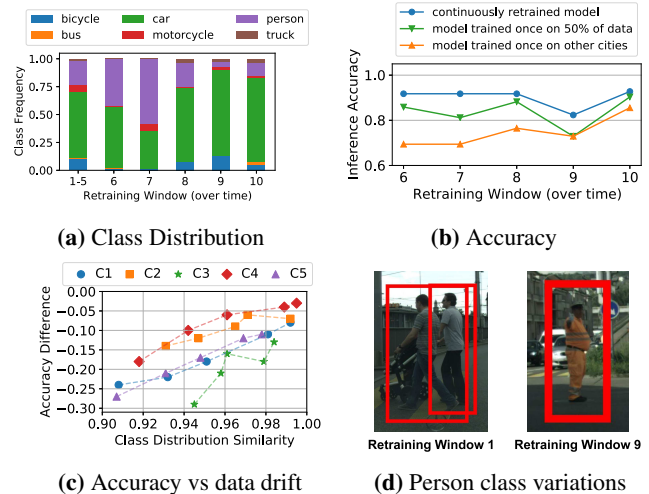
Figure 2: Continuous learning in the Cityscapes dataset. Shift in class distributions (a) across windows necessitates continuous learning (b). Model accuracy is not only affected by class distribution shifts (c), but also by changes in object appearances (d).

low-cost "student" model with a high-cost "teacher" model (or knowledge distillation [33]), and this has been broadly applied in computer vision literature [42, 72, 93, 96].

2.3 Accuracy benefits of continuous learning

To show the benefits of continuous learning, we use the video stream from one example city in the Cityscapes dataset [57] that consists of videos from dashboard cameras in many cities. In our evaluation in §6, we use both moving dashboard cameras as well as static cameras over long time periods. We divide the video data in our example city into ten fixed *retraining windows* (200s in this example).

Understanding sources of data drift. Figure 2a shows the change of object class distributions across windows. The initial five windows see a fair amount of persons and bicycles, but bicycles rarely show up in windows 6 and 7, while the share of persons varies considerably across windows 6 − 10. Figure 2c summarizes the effect of this data drift on model accuracy in five independent video streams, C1-C5. For each stream, we train a baseline model on the first five windows, and test it against five windows in the future and use cosine similarity to measure the class distribution shift for each window. Though accuracy generally improves when the model is used on windows with similar class distributions (high cosine similarity), the relationship is not guaranteed (C2, C3). This is because class distribution shift is not the only form of data drift. Illumination, pose and appearance differences also affect model performance (e.g. clothing and angles for objects in the person class vary significantly; Figure 2d).

Improving accuracy with continuous learning. Figure 2b plots inference accuracy of an edge DNN (a compressed ResNet18 classifier) in the last five windows using different training options. (1) Training a compressed ResNet18 with video data on all other cities of the Cityscapes dataset does not
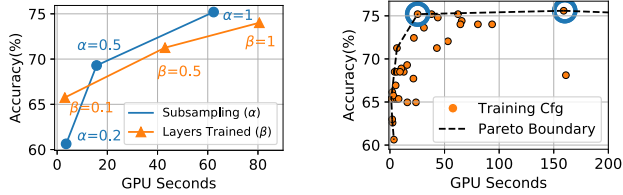
(a) Effect of Hyperparameters   **(b)** Resource-accuracy

Figure 3: Measuring retraining configurations. GPU seconds refers to the duration taken for retraining with 100% **GPU allocation. (a) varies two example hyperparameters, keeping others constant. Note the Pareto boundary of configurations in (b); for every non-Pareto configuration, there is at least one Pareto configuration that is better than it in *both* accuracy and GPU cost.**

result in good performance. (2) Unsurprisingly, we observe that training the edge DNN once using data from the first five windows *of this example city* improves the accuracy. (3) *Continuous retraining* using the most recent data for training achieves the highest accuracy consistently. Its accuracy is higher than the other options by up to 22%.

Interestingly, using the data from the first five windows to train the larger ResNet101 DNN (not graphed) achieves better accuracy than the continuously retrained ResNet18. The substantially better accuracy of ResNet101 compared to ResNet18 when trained *on the same data* of the first five windows also shows that this training data was indeed fairly representative. But the lightweight ResNet18's weights and architecture limits its ability to learn and is a key contributor to its lower accuracy. Nonetheless, ResNet101 is 13× slower than the compressed ResNet18 [21]. This makes the efficient ResNet18 more suited for edge deployments and continuous learning enables it to maintain high accuracy even with data drift. Therefore, the need for continuous training of edge DNNs is ongoing and not just during a "ramp-up" phase.

3 Scheduling retraining and inference jointly

We propose *joint retraining and inference* on edge servers. The joint approach utilizes resources better than statically provisioning compute for retraining. Since retraining is periodic [72, 93] with far higher compute demands than inference, static provisioning causes idling. Compared to uploading videos to the cloud for retraining, our approach has advantages in privacy (§2.1), and network costs and accuracy (§6.4).

3.1 Configuration diversity of retraining and inference

Tradeoffs in retraining configurations. The hyperparameters for retraining, or "retraining configurations", influence the resource demands and accuracy. Retraining fewer layers of the DNN (or, "freezing" more layers) consumes lesser GPU resources, as does training on fewer data samples, but they also produce a model with lower accuracy; Figure 3a.

Figure 3b illustrates the resource-accuracy trade-offs for an edge DNN (ResNet18) with various hyperparameters: number of training epochs, batch sizes, number of neurons in the last

Configuration	Retraining Window 1		Retraining Window 2	
	End Accuracy	GPU seconds	End Accuracy	GPU seconds
Video A Cfg1A	75	85	95	90
Video A Cfg2A (*)	70	65	90	40
Video B Cfg1B	90	80	98	80
Video B Cfg2B (*)	85	50	90	70

Table 1: Hyperparameter configurations for retraining jobs in Figure 4's example. At the start of retraining window 1, camera A's inference model has an accuracy of 65% and camera B's inference model has an accuracy of 50%. Asterisk (*) denotes the configurations picked in Figures 4b and 4d.

layer, number of frozen layers, and fraction of training data. We make two observations. First, there is a wide spread in the resource usage (measured in GPU seconds), by upto a factor of 200×. Second, higher resource usage does not always yield higher accuracy. For the two configurations circled in Figure 3b, their GPU demands vary by 6× even though their accuracies are the same (∼ 76%). Thus, careful selection of the configurations considerably impacts the resource efficiency. Moreover, the accuracy spread across configurations is dependent on the extent of data-drift. Retraining on visually similar data with little drift results in a narrower spread. With the changing characteristics of videos, it is challenging to efficiently obtain the resource-accuracy profiles for retraining.

Tradeoffs in inference configurations. Inference pipelines also allow for flexibility in their resource demands at the cost of accuracy through configurations to downsize and sample frames [59]. Reducing the resource allocation to inference pipelines increases the processing latency per frame, which then calls for sub-sampling the incoming frames to match the processing rate, that in turn reduces inference accuracy [32]. Prior work has made dramatic advancements in profilers that efficiently obtain the resource-accuracy relationship for *inference configurations* [37]. We use these efficient inference profilers in our solution, and also to ensure that the inference pipelines keep up with analyzing the live video streams.

3.2 Illustrative scheduling example

We use an example with 3 GPUs and two video streams, A and B, to show the considerations in scheduling inference and retraining tasks jointly. Each retraining uses data samples accumulated since the *beginning of* the last retraining (referred to as the "retraining window").[2] To simplify the example, we assume the scheduler has knowledge of the resource-accuracy profiles, but these are expensive to get in practice (we describe our efficient solution for profiling in §4.3). Table 1 shows the retraining configurations (Cfg1A, Cfg2A, Cgf1B, and Cgf2B), their respective accuracies after the retraining, and GPU cost.

---

[2]Continuous learning targets retraining windows of tens of seconds to few minutes [72, 93]. We use 120 seconds in this example. Our solution is robust to and works with any given window duration for its decisions (See §6.2).

(a) Uniform scheduler   **(b)** Accuracy-optimal sched.

(c) Uniform scheduler   **(d)** Accuracy-optimal sched.
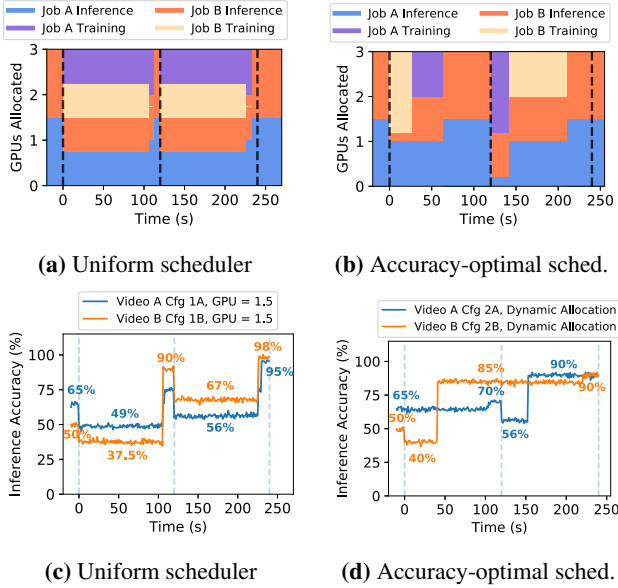
Figure 4: Resource allocations (top) and inference accuracies (bottom) over time for two retraining windows (each of 120s). The left figures show a uniform scheduler which evenly splits the 3 GPUs, and picks configurations resulting in the most accurate models. The right figures show the accuracy-optimized scheduler that prioritizes resources and optimizes for inference accuracy averaged over the retraining window (73% compared to the uniform scheduler's 56%). The accuracy-optimized scheduler also ensures that inference accuracy never drops below a minimum (set to 40% in this example, denoted as a_{MIN}).

The scheduler is responsible for selecting configurations and allocating resources for inference and retraining jobs.

Uniform scheduling: Building upon prior work in cluster schedulers [9, 80] and video analytics systems [32], a baseline solution for resource allocation evenly splits the GPUs between video streams, and each stream evenly partitions its allocated GPUs for retraining and inference tasks; see Figure 4a. Just like model training systems [29, 44, 45], the baseline always picks the configuration for retraining that results in the highest accuracy (Cfg1A, Cfg1B for both windows).

Figure 4c shows the *inference* accuracies of the two live streams. We see that when the retraining tasks take resources away from the inference tasks, the inference accuracy drops significantly (65% → 49% for video A and 50% → 37.5% for video B in Window 1). While the inference accuracy increases *after* retraining, it leaves too little time in the window to reap the benefit of retraining. Averaged across both retraining windows, the inference accuracy across the two video streams is only 56% because the gains due to the improved accuracy of the retrained model are undercut by the time taken for retraining (during which inference accuracy suffered).

Accuracy-optimized scheduling: Figures 4b and 4d illustrate an accuracy-optimized scheduler, which by taking a holistic view on the multi-dimensional tradeoffs, provides an an average inference accuracy of 73%. In fact, to match the accura-

cies, the above uniform scheduler would require nearly twice the GPUs (i.e., 6 GPUs instead of 3 GPUs).

This scheduler makes three key improvements. First, the scheduler selects the hyperparameter configurations based on their accuracy improvements *relative* to their GPU cost. It selects lower accuracy options (Cfg2A/Cfg2B) instead of the higher accuracy ones (Cfg1A/Cfg1B) because these configurations are substantially cheaper (Table 1). Second, the scheduler *prioritizes* retraining tasks that yield higher accuracy, so there is more time to reap the benefit from retraining. For example, the scheduler prioritizes B's retraining in Window 1 as its inference accuracy after retraining increases by 35% (compared to 5% for video A). Third, the scheduler controls the accuracy drops during retraining by balancing the retraining time and the resources taken away from inference.

4 Ekya: Solution Description

Continuous training on limited edge resources requires smartly deciding when to retrain each video stream's model, how much resources to allocate, and what configurations to use. Making these decisions presents two challenges.

First, the decision space of multi-dimensional configurations and resource allocations is computationally more complex than two fundamentally challenging problems of multi-dimensional knapsack and multi-armed bandits (§4.1). Hence, we design a **thief scheduler** (§4.2), a heuristic that makes the joint retraining-inference scheduling tractable in practice.

Second, the scheduler requires the model's exact performance (in resource usage and inference accuracy), but this requires retraining using all the configurations. We address this challenge with our **micro-profiler** (§4.3), which retrains only a few select configurations on a fraction of the data. Figure 5 presents an overview of Ekya's components.

4.1 Formulation of joint inference and retraining

The problem of joint inference and retraining aims to maximize overall inference accuracy for all video streams $\mathcal{V}$ in a retraining window T with duration $\|T\|$. All work must be done in $\mathcal{G}$ GPUs. Thus, the total compute capability is $\mathcal{G}\|T\|$ GPU-time. Without loss of generality, let δ be the smallest granularity of GPU allocation. Each video $v \in V$ has a set of *retraining* configurations Γ and a set of *inference* configurations Λ (§3.1). Table 4 (§A) lists the notations.

Decisions. For each video $v \in \mathcal{V}$ in a window T, we decide: (1) the retraining configuration $\gamma \in \Gamma$ ($\gamma = \emptyset$ means no retraining); (2) the inference configuration $\lambda \in \Lambda$; and (3) how many GPUs (in multiples of δ) to allocate for retraining ($\mathcal{R}$) and inference (I). We use binary variables $\phi_{v\gamma\lambda\mathcal{R}I} \in \{0, 1\}$ to denote these decisions (see Table 4 §A for the definition). These decisions require $C_T(v, \gamma, \lambda)$ GPU-time and yields overall accuracy of $A_T(v, \gamma, \lambda, \mathcal{R}, I)$. $A_T(v, \gamma, \lambda, \mathcal{R}, I)$ is averaged across the window T (§3.2), and the above decisions determine the inference accuracy at *each point in time*.

Optimization. Maximize the inference accuracy averaged
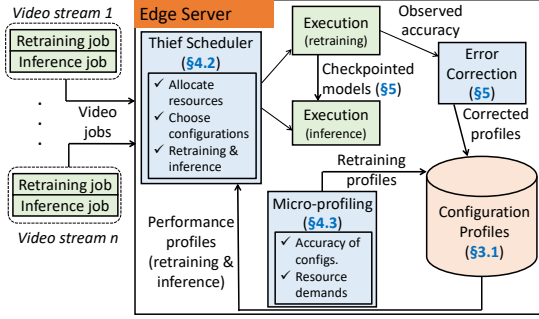
Figure 5: Ekya's components and their interactions.

across all videos in a retraining window within the GPU limit.

$$\underset{\phi_{v\gamma\lambda\mathcal{R}I}}{\arg\max}\frac{1}{\|\mathcal{V}\|}\sum_{\substack{\forall v\in\mathcal{V},\forall\gamma\in\Gamma,\forall\lambda\in\Lambda,\\ \forall\mathcal{R},\forall I\in\{0,1,...,\frac{G}{\delta}\}}}\phi_{v\gamma\lambda\mathcal{R}I}\cdot A_T(v,\gamma,\lambda,\mathcal{R},I)$$

subject to

1. $\displaystyle\sum_{\substack{\forall v\in\mathcal{V},\forall\gamma\in\Gamma,\forall\lambda\in\Lambda,\\ \forall\mathcal{R},\forall I}}\phi_{v\gamma\lambda\mathcal{R}I}\cdot C_T(v,\gamma,\lambda)\leq G\|T\|$

2. $\displaystyle\sum_{\substack{\forall v\in\mathcal{V},\forall\gamma\in\Gamma,\forall\lambda\in\Lambda,\\ \forall\mathcal{R},\forall I}}\phi_{v\gamma\lambda\mathcal{R}I}\cdot(\mathcal{R}+I)\leq\frac{G}{\delta}$ (1)

3. $\displaystyle\sum_{\substack{\forall\gamma\in\Gamma,\forall\lambda\in\Lambda,\\ \forall\mathcal{R},\forall I}}\phi_{v\gamma\lambda\mathcal{R}I}\leq 1,\forall v\in\mathcal{V}$

The first constraint ensures that the GPU allocation does not exceed the available GPU-time $G\|T\|$ in the retraining window. The second constraint limits the *instantaneous* allocation (in multiples of δ) to never exceed the available GPUs. The third constraint ensures that at most one configuration is picked for retraining and inference each for a video v.

Our analysis in §A.1 shows that the above optimization problem is *harder* than the multi-dimensional binary knapsack problem and modeling the uncertainty of $A_T(v,\gamma,\lambda,\mathcal{R},I)$ is more challenging than the multi-armed bandit problem.

4.2 Thief Scheduler

Our scheduling heuristic makes the scheduling problem tractable by decoupling resource allocation (i.e., $\mathcal{R}$ and I) and configuration selection (i.e., γ and λ) (Algorithm 1). We refer to Ekya's scheduler as the "thief" scheduler and it iterates among all inference and retraining jobs as follows.

(1) It starts with a fair allocation for all video streams $v\in V$ (line 2 in Algorithm 1). In each step, it iterates over all the inference and retraining jobs of each video stream (lines 5-6), and *steals* a tiny quantum Δ of resources (in multiples of δ; see Table 4, §A) from each of the other jobs (lines 10-11).

(2) With the new resource allocations (temp_alloc[]), it then selects configurations for the jobs using the PickConfigs method (line 14 and Algorithm 2, §A) that iterates over all the configurations for inference and retraining for each video stream. For inference jobs, among all the configurations whose accuracy is $\geq a_{\mathrm{MIN}}$, PickConfigs picks the configuration with the highest

accuracy that can keep up with the inference of the live video stream given current allocation (line 3-4, Algorithm 2, §A).

For retraining jobs, PickConfigs picks the configuration that maximizes the accuracy $A_T(v,\gamma,\lambda,\mathcal{R},I)$ over the retraining window for each video v (lines 6-12, Algorithm 2, §A). EstimateAccuracy (line 7, Algorithm 2, §A) aggregates the instantaneous accuracies over the retraining window for a given pair of inference configuration (chosen above) and retraining configuration. Ekya's micro-profiler (§4.3) provides the estimate of the accuracy and the time to retrain for a retraining configuration when 100% of GPU is allocated, and EstimateAccuracy proportionally scales the GPU-time for the current allocation (in temp_alloc[]) and training data size. In doing so, it avoids configurations whose retraining durations exceed $\|T\|$ with the current allocation (first constraint in Eq. 1).

(3) After reassigning the configurations, Ekya uses the estimated average inference accuracy (accuracy_avg) over the retraining window (line 14 in Algorithm 1) and keeps the new allocations only if it improves up on the accuracy from prior to stealing the resources (line 15 in Algorithm 1).

The thief scheduler repeats the process till the accuracy stops increasing (lines 15-20 in Algorithm 1) and until all the jobs have played the "thief". Algorithm 1 is invoked at the beginning of each retraining window, as well as on the completion of every training job during the window.

Design rationale: We call out the key aspects that makes the scheduler's decision efficient by pruning the search space.

- *Coarse allocations:* The thief scheduler allocates GPU resources in quantums of Δ. Intuitively, Δ is the step size for allocation used by the scheduler. Thus, the final resource allocation from the thief scheduler is within Δ of the optimal allocation. We empirically pick a Δ that is coarse yet accurate enough in practice, while being mindful of modern GPUs[4]; see §6.2. Algorithm 1 ensures that the total allocation is within the limit (second constraint in Eq 1).

- *Reallocating resources only when a retraining job completes:* Although one can reallocate GPU resource among jobs at finer temporal granularity (*e.g.,* whenever a retraining job has reached a high accuracy), we empirically find that the gains from such complexity is marginal.

- *Pruned configuration list:* Our micro-profiler (described next) speeds up the thief scheduler by giving it only the more promising configurations. Thus, the list Γ used in Algorithm 1 is significantly smaller than the exhaustive set.

4.3 Performance estimation with micro-profiling

Ekya's scheduling decisions in §4.2 rely on estimations of post-retraining accuracy and resource demand of the retraining configurations. Specifically, at the beginning of each retraining window T, we need to *profile* for each video v and each configuration $\gamma\in\Gamma$, the accuracy after retraining using γ and the corresponding time taken to retrain.

Profiling in Ekya vs. hyperparameter tuning: While Ekya's profiling may look similar to hyperparameter tuning

Algorithm 1: Thief Scheduler.

Data: Training (Γ) and inference (Λ) configurations
Result: GPU allocations $\mathcal{R}$ and I, chosen configurations
$\quad\quad\quad (\gamma \in \Gamma, \lambda \in \Lambda) \; \forall v \in V$

```
1  all_jobs[] = Union of inference and training jobs of videos V;
   /* Initialize with fair allocation            */
2  best_alloc[] = fair_allocation(all_jobs);
3  best_configs[], best_accuracy_avg = PickConfigs(best_alloc);
   /* Thief resource stealing                     */
4  for thief_job in all_jobs[] do
5      for victim_job in all_jobs[] do
6          if thief_job == victim_job then continue;
7          temp_alloc[] ← best_alloc[];
8          while true do
               /* Δ is the increment of stealing    */
9              temp_alloc[victim_job] −= Δ;
10             temp_alloc[thief_job] += Δ;
11             if temp_alloc[victim_job] < 0 then  break ;
               /* Calculate accuracy over retraining
                  window and pick configurations.   */
12             temp_configs[], accuracy_avg =
                 PickConfigs(temp_alloc[]);
13             if accuracy_avg > best_accuracy_avg then
14                 best_alloc[] = temp_alloc[];
15                 best_accuracy_avg = accuracy_avg;
16                 best_configs[] = temp_configs[];
17             else
18                 break;

19 return best_alloc[], best_configs[];
```

(e.g., [46, 48, 62, 85]) at first blush, there are two key differences. First, Ekya needs the performance estimates of a broad set of candidate configurations for the thief scheduler, not just of the single best configuration, because the best configuration is jointly decided across the many retraining and inference jobs. Second, in contrast to hyperparameter tuning which runs separately of the eventual inference/training, Ekya's profiling must share compute resource with all retraining and inference.

Opportunities: Ekya leverages three empirical observations for efficient profiling of the retraining configurations. (*i*) Resource demands of the configurations are deterministic. Hence, we measure the GPU-time taken to retrain for *each epoch* in the current retraining window when 100% of the GPU is allocated to the retraining. This GPU-time must then be re-scaled for varying number of epochs, GPU allocations, and training data sizes in Algorithm 1. For re-scaling number of epochs and training data sizes, we linearly scale the GPU-time. For re-scaling GPU allocations, we use an offline computed profile of the model throughput for different resource allocations to account for sub-linear scaling. Our real testbed-based evaluation shows that these rescaling functions works well in practice. (*ii*) Post-retraining accuracy can be roughly estimated by training on a small subset of training data for a handful of epochs. (*iii*) The thief scheduler's deci-

sions are not impacted by small errors in the estimations.

Micro-profiling design: The above insights inspired our approach, called *micro-profiling*, where for each video, we test the retraining configurations on a *small subset* of the retraining data and only for a *small number* of epochs (well before models converge). Our micro-profiler is $100\times$ more efficient than exhaustive profiling (of all configurations on the entire training data), while predicting accuracies with an error of 5.8%, which is low enough in practice to *mostly* ensure that the thief scheduler makes the same decisions as it would with a fully accurate prediction. We use these insights to now explain the techniques that make Ekya's micro-profiling efficient.

1) Training data sampling: Ekya's micro-profiling works on only a small fraction (say, $5\% - 10\%$) of the training data in the retraining window (which is already a subset of all the videos accumulated in the retraining window). While we considered weighted sampling techniques for the micro-profiling, we find that uniform random sampling is the most indicative of the configuration's performance on the full training data, since it preserves all the data distributions and variations.

2) Early termination: Similar to data sampling, Ekya's micro-profiling only tests each configuration for a small number (say, 5) of training epochs. Compared to a full fledged profiling that needs few tens of epochs to converge, such early termination greatly speeds up the micro-profiling process.

After early termination on the sampled training data, we obtain the (validation) accuracy of each configuration at each epoch it was trained. We then fit the accuracy-epoch points to the a non-linear curve model from [70] using a non-negative least squares solver [6]. This model is then used to extrapolate the accuracy that would be obtained by retraining with all the data for larger number of epochs. The use of this extrapolation is consistent with similar work in this space [55, 70].

3) Pruning bad configurations: Finally, Ekya's micro-profiling also prunes out those configurations for micro-profiling (and hence, for retraining) that have historically not been useful. These are configurations that are significantly distant from the configurations on the Pareto curve of the resource-accuracy profile (see Figure 3b), and thus unlikely to be picked by the thief scheduler. To bootstrap pruning, all configurations are evaluated in the first window. After every 2 windows, a fixed fraction of the worst performing configurations are dropped. While first few retraining windows must explore a big space of configurations, the search space size drops exponentially over time. Avoiding these configurations improves the efficiency of the micro-profiling.

Annotating training data: For both the micro-profiling as well as the retraining, Ekya acquires labels using a "golden model" (§2.2). This is a high-cost but high-accuracy model trained on a large dataset. As explained in §2, the golden model cannot keep up with inference on the live videos and we use it to label only a small subset of the videos for retraining. The delay of annotating training data with the golden model

is accounted by the scheduler as follows: we subtract the data annotation delay from the retraining window and only pass the remaining time of the window to Algorithm 2 (§A).

5 Implementation and Experimental Setup

Implementation: Ekya uses PyTorch [66] for running and training ML models, and each component is implemented as a collection of long-running processes with the Ray[63] actor model. The micro-profiler and training/inference jobs run as independent actors which are controlled by the thief scheduler actor. Ekya achieves fine-grained and dynamic reallocation of GPU between training and inference processes using Nvidia MPS [4], which provides resource isolation within a GPU by intercepting CUDA calls and rescheduling them. Our implementation also adapts to errors in profiling by reactively adjusting its allocations if the actual model performance diverges from the predictions of the micro-profiler. Ekya's code and datasets are available at the project page: aka.ms/ekya

Datasets: We use both on-road videos captured by dashboard cameras as well as urban videos captured by mounted cameras. The dashboard camera videos are from cars driving through cities in the US and Europe, Waymo Open [68] (1000 video segments with in total 200K frames) and Cityscapes [57] (5K frames captured by 27 cameras) videos. The urban videos are from stationary cameras mounted in a building ("Urban Building") as well as from five traffic intersections ("Urban Traffic"), both collected over 24-hour durations. We use a retraining window of 200 seconds in our experiments, and split each of the videos into 200 second segments. Since the Waymo and Cityscapes dataset do not contain continuous timestamps, we create retraining windows by concatenating images from the same camera in chronological order to form a long video stream and split it into 200 second segments.

DNNs: We demonstrate Ekya's effectiveness on two machine learning tasks – object classification and object detection – using multiple compressed edge DNNs for each task: (*i*) object classification using ResNet18[39], MobileNetV2[53] and ShuffleNet[98], and (*ii*) object detection using TinyYOLOv3[75] and SSD[49]. As explained in §2.2, we use an expensive golden model (ResNeXt 101 [91] for object classification and YOLOv3 [75] for object detection) to get ground truth labels for training and testing.

Testbed and trace-driven simulator: We run Ekya's implementation on AWS EC2 p3.2xlarge instances for 1 GPU experiments and p3.8xlarge for 2 GPU experiments. Each instance has Nvidia V100 GPUs with NVLink interconnects.

We also built a simulator to test Ekya under a wide range of resource constraints, workloads, and longer durations. The simulator takes as input the accuracy and resource usage (in GPU time) of training/inference configurations logged from our testbed. For each training job, we log the accuracy over GPU-time. We also log the inference accuracy on the real videos. This exhaustive trace allows us to mimic the jobs with high fidelity under different scheduling policies.

Retraining configurations: Our retraining configurations combine the number of epochs to train, batch size, number of neurons in the last layer, number of layers to retrain, and the fraction of data between retraining windows to use for retraining (§3.1). For the object detection models (TinyYOLO and SSDLite), we set the batch size to 8 and the fraction of layers frozen between 0.7 and 0.9. The resource requirements of the configurations for the detection models vary by $153\times$.

Baselines: Our baseline, called *uniform scheduler*, uses (*a*) a fixed retraining configuration, and (*b*) a static retraining/inference resource allocation (these are adopted by prior schedulers [9, 32, 80]). For each dataset, we test all retraining configurations on a hold-out dataset [3] (*i.e.,* two video streams that were never used in later tests) to produce the Pareto frontier of the accuracy-resource tradeoffs (*e.g.,* Figure 3). The uniform scheduler then picks two points on the Pareto frontier as the fixed retraining configurations to represent "high" (Config 1) and "low" (Config 2) resource usage, and uses one of them for all retraining windows in a test.

We also consider two alternatives in §6.4. (1) *offloading retraining to the cloud*, and (2) *caching and re-using a retrained model* from history based on various similarity metrics.

6 Evaluation

We evaluate Ekya's performance, and the key findings are:
1) Compared to static retraining baselines, Ekya achieves upto 29% higher accuracy for compressed vision models in both classification and detection. For the baseline to match Ekya's accuracy, it would need $4\times$ additional GPU resources. (§6.1)
2) Both micro-profiling and thief scheduler contribute sizably to Ekya's gains. (§6.2) In particular, the micro-profiler estimates accuracy with low median errors of 5.8%. (§6.3)
3) The thief scheduler efficiently makes its decisions in 9.4s when deciding for 10 video streams across 8 GPUs with 18 configurations per model for a 200s retraining window. (§6.2)
4) Compared to alternate designs, including reusing cached history models trained on similar data/scenarios as well as retraining the models in the cloud, Ekya achieves significantly higher accuracy without the network costs (§6.4).

6.1 Overall improvements

We evaluate Ekya and the baselines along three dimensions— *inference accuracy* (% of images correctly classified for object classification, F1 score (measured at a 0.3 threshold for the Intersection-over-Union of the bounding box) for detection), *resource consumption* (in GPU time), and *capacity* (the number of concurrently processed video streams). Note that the evaluation always keeps up with the video frame rate (*i.e.,* no indefinite frame queueing). By default we evaluate the performance of Ekya on ResNet18 models, but we also show that it generalizes to other model types and vision tasks.

---

[3] The same hold-out dataset is used to customize the off-the-shelf DNN inference model. This is a common strategy in prior work (*e.g.,* [22]).
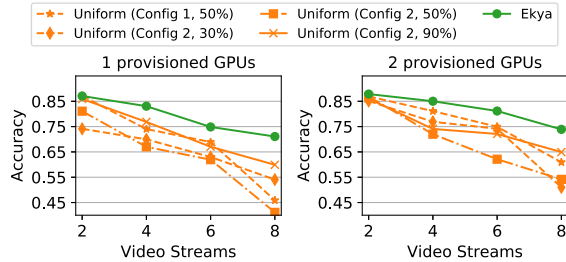
(a) Cityscapes
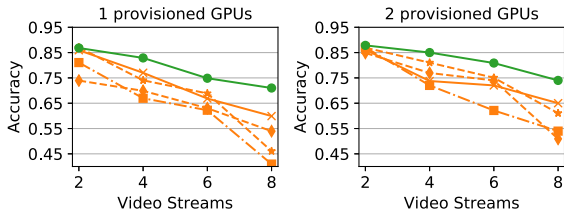
(b) Waymo

Figure 6: **Effect of adding video streams on accuracy with different schedulers. When more video streams share resources, Ekya's accuracy gracefully degrades while the baselines' accuracy drops faster. ("Uniform (Cfg 1, 90%)" means the uniform scheduler allocates 90% GPU to inference, 10% to retraining)**

Accuracy vs. Number of concurrent video streams: Figure 6 shows the ResNet18 model's accuracy with Ekya and the baselines when analyzing a growing number of concurrent video streams under a fixed number of provisioned GPUs for Waymo and Cityscapes datasets. The uniform baselines use different combinations of pre-determined retraining configurations and resource partitionings. For consistency, the video streams are shuffled and assigned an id (0-10), and are then introduced in the same increasing order of id in all experiments. This ensures that different schedulers tested for k parallel streams use the same k streams, and these k streams are always a part of any k' streams ($k' > k$) used for testing.

As the number of video streams increases, Ekya enjoys a growing advantage (upto 29% under 1 GPU and 23% under 2 GPU) in accuracy over the uniform baselines. This is because Ekya gradually shifts more resource from retraining to inference and uses cheaper retraining configurations. In contrast, increasing the number of streams forces the uniform baseline to allocate less GPU cycles to each inference job, while retraining jobs, which use fixed configurations, slow down and take the bulk of each window.

Generalizing to other ML models: Ekya's thief scheduler can be readily applied to any ML model and task (e.g., classification or detection) that needs to be fine-tuned continuously on newer data. To demonstrate this, we evaluate Ekya with:

- *Other object classifiers:* Figure 7a shows the performance of Ekya when running MobileNetV2 and ShuffleNet as the edge models in two independent setups for object classification at the edge. Continuing the trend that we observed for ResNet18 (in Figure 6), Figure 7a shows that Ekya leads

Scheduler	Capacity		Scaling factor
	1 GPU	2 GPUs	
Ekya	**2**	**8**	**4x**
Uniform (Config 1, 50%)	2	2	1x
Uniform (Config 2, 90%)	2	4	2x
Uniform (Config 2, 50%)	2	4	2x
Uniform (Config 2, 30%)	0	2	-

Table 2: **Capacity (number of video streams that can be concurrently supported subject to accuracy target 0.75) vs. number of provisioned GPUs. Ekya scales better than the uniform baselines with more available compute resource.**
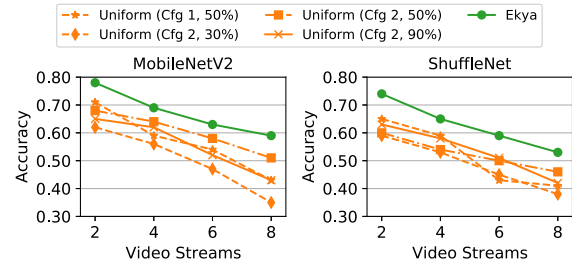
(a) Generalize across object classification models
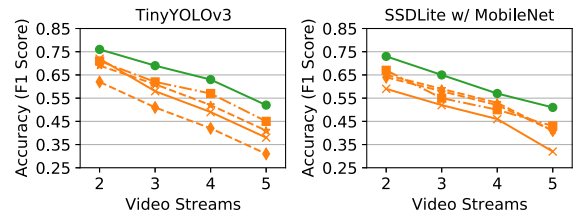
(b) Object Detection Models

Figure 7: **Improvement of Ekya extends to two more compressed DNN classifiers and two popular object detectors.**

to up to 22% better accuracy than uniform baselines.

- *Object detection models:* In addition to object classification, we also evaluate using object detection tasks which detect the bounding boxes of objects in the video stream. Figure 7b shows Ekya outperforms the uniform baseline's F1 score by 19% when processing same number of concurrent video streams. Importantly, Ekya's design broadly applies to new tasks without any systemic changes.

These gains stem from Ekya's ability to navigate the rich resource-accuracy space of models by carefully selecting training and inference hyperparameters (e.g., the width multiplier in MobileNetV2, convolution sparsity in ShuffleNet). For the rest of our evaluation, we only present results with ResNet18 though the observations hold for other models.

Number of video streams vs. provisioned resource: We compare Ekya's *capacity* (defined by the maximum number of concurrent video streams subject to an accuracy threshold) with that of uniform baseline, as more GPUs are available. Setting an accuracy threshold is common in practice, since applications usually require accuracy to be above a threshold for the inference to be usable. Table 2 uses the Cityscapes results (Figure 6) to derive the scaling factor of capacity vs.

(a) Cityscapes

(b) Waymo

(c) Urban Building

(d) Urban Traffic
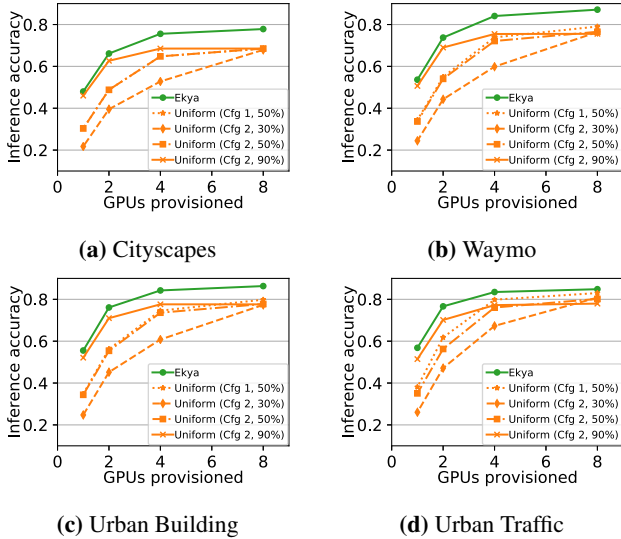
Figure 8: Inference accuracy of different schedulers when processing 10 video streams under varying GPU provisionings.

(a) Video stream #1
(Inference accuracy = 0.82)

(b) Video stream #2
(Inference accuracy = 0.83)
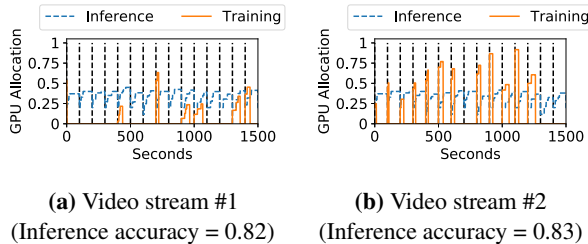
Figure 9: Ekya's resource allocation to two video streams over time. Ekya adapts when to retrain each stream's model and allocates resource based on the retraining benefit to each stream.

the number of provisioned GPUs and shows that with more provisioned GPUs, Ekya scales faster than uniform baselines.

Accuracy vs. provisioned resource: Finally, Figure 8 stress-tests Ekya and the uniform baselines to process 10 concurrent video streams and shows their average inference accuracy under different number of GPUs. To scale to more GPUs, we use the simulator (§5), which uses profiles recorded from real tests and we verified that it produced similar results as the implementation at small-scale. As we increase the number of provisioned GPUs, we see that Ekya consistently outperforms the best of the two baselines by a considerable margin and more importantly, with 4 GPUs Ekya achieves higher accuracy (marked with the dotted horizontal line) than the baselines at 16 GPUs (*i.e.,* 4× resource saving).

The above results show that Ekya is more beneficial when there is high contention for the GPU on the edge. Under low contention, the room for improvement shrinks. Contention is, however, common in the edge since the resources are tightly provisioned to minimize their idling.

6.2 Understanding Ekya's improvements

Resource allocation across streams: Figure 9 shows Ekya's resource allocation across two example video streams over

several retraining windows. In contrast to the uniform baselines that use the same retraining configuration and allocate equal resource to retraining and inference (when retraining takes place), Ekya retrains the model only when it benefits and allocates different amounts of GPUs to the retraining jobs of video streams, depending on how much accuracy gain is expected from retraining on each stream. In this case, more resource is diverted to video stream #1 (#1 can benefit more from retraining than #2) and both video streams achieve much higher accuracies (0.82 and 0.83) than the uniform baseline.

Component-wise contribution: Figure 10a understands the contributions of resource allocation and configuration selection (on 10 video streams with 4 GPUs provisioned). We construct two variants from Ekya: *Ekya-FixedRes*, which removes the smart resource allocation in Ekya (*i.e.,* using the inference/training resource partition of the uniform baseline), and *Ekya-FixedConfig* removes the microprofiling-based configuration selection in Ekya (*i.e.,* using the fixed configuration of the uniform baseline). Figure 10a shows that both adaptive resource allocation and configuration selection has a substantial contribution to Ekya's gains in accuracy, especially when constrained (i.e., fewer resources are provisioned).

Retraining window sensitivity analysis: Figure 10b evaluates the sensitivity of Ekya to the retraining window size. Ekya is robust to different retraining window sizes. When the retraining window size is too small (10 seconds), the accuracy of Ekya is equivalent to no retraining accuracy due to insufficient time and resources for retraining. As the window increases, Ekya's performance quickly ramps up because the thief scheduler is able to allocate resources to retraining. As the retraining window size further increases Ekya's performance slowly starts moderately degrading because of the inherent limitation in capacity of compressed models (§2.3).

Impact of scheduling granularity: A key parameter in Ekya's scheduling algorithm (§4.2) is the allocation quantum Δ: it controls the runtime of the scheduling algorithm and the granularity of resource allocation. In our sensitivity analysis with 10 video streams, we see that increasing Δ from 1.0 (coarse-grained; one full GPU) to 0.1 (fine-grained; fraction of a GPU), increases the accuracy substantially by $\sim 8\%$. Though the runtime also increases to 9.5 seconds, it is still a tiny fraction (4.7%) of the retraining window (200s).

6.3 Effectiveness of micro-profiling

The absolute cost of micro-profiling is small; for our experiments, micro-profiling takes 4.4 seconds for a 200s window.

Errors of microprofiled accuracy estimates: Ekya's microprofiler estimates the accuracy of each configuration (§4.3) by training it on a subset of the data for a small number of epochs. To evaluate the micro-profiler's estimates, we run it on all configurations for 5 epochs and on 10% of the retraining data from all streams of the Cityscapes dataset, and calculate the estimation error against the retrained accuracies when trained

(a) Factor analysis **(b)** Sensitivity to retraining window size.
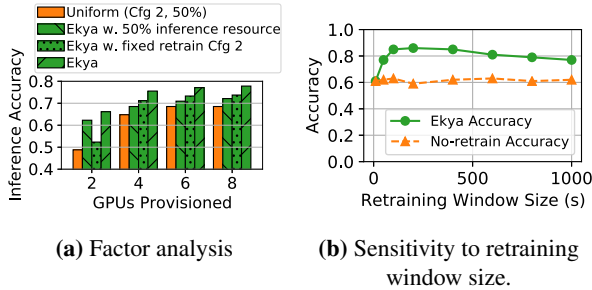
Figure 10: **(a) Component-wise impact of removing dynamic resource allocation (50% allocation) or removing retraining configuration adaptation (fixed Cfg 2). (b) Robustness of Ekya to a wide range of retraining window values.**

(a) Distribution of accuracy estimation errors. **(b)** Impact of an controlled error ε to accuracy estimates.
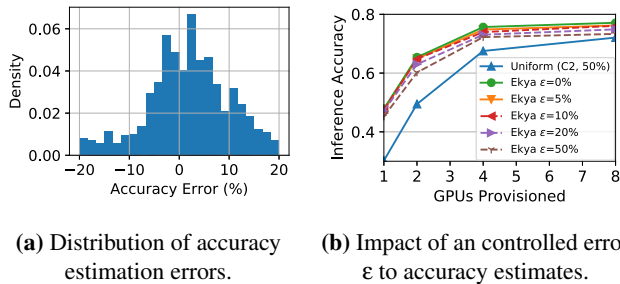
Figure 11: **Evaluation of microprofiling performance. (a) shows the distribution of microprofiling's actual estimation errors, and (b) shows the robustness of Ekya's performance against microprofiling's estimation errors.**

on 100% of the data for 5, 15 and 30 epochs. Figure 11a plots the distribution of the errors in accuracy estimation and and show that the micro-profiled estimates are largely unbiased with an median absolute error of 5.8%.

Sensitivity to microprofiling estimation errors: Finally, we test the impact of accuracy estimation errors (§4.3) on Ekya. We add gaussian noise on top of the predicted retraining accuracy when the microprofiler is queried. Figure 11b shows that Ekya is robust to accuracy estimate errors: with upto 20% error (which covers all errors in Figure 11a) in the profiler prediction, the maximum accuracy drop is 3%.

6.4 Comparison with alternative designs

Ekya vs. Cloud-based retraining: One may upload a sub-sampled video stream to the cloud, retrain the model, and download the model back to the edge [40]. While this solution is not an option for many deployments due to legal and privacy stipulations [11, 87], we still evaluate this option as it lets the edge servers focus on inference. Cloud-based solutions, however, results in lower accuracy due to significant network delays on the constrained networks typical of edges [81].

For example, consider 8 video streams running ResNet18 and a retraining window of 400 seconds. A HD (720p) video stream at 4Mbps and 10% data sub-sampling (typical in our experiments) amounts to 160Mb of training data per camera per window. Uploading 160Mb for each of the 8 cameras over

	Bandwidth (Mbps)		Acc.	Bandwidth Gap	
	Uplink	Downlink		Uplink	Downlink
Cellular	5.1	17.5	68.5%	10.2×	3.8×
Satellite	8.5	15	69.2%	5.9×	4.4×
Cellular (2×)	10.2	35	71.2%	5.1×	1.9×
Ekya	-	-	**77.8%**	-	-

Table 3: **Retraining in the cloud under different networks [58, 65, 81] versus using Ekya at the edge. Ekya achieves better accuracy without using expensive satellite and cellular links.**

(a) Ekya vs. re-using cached models over time **(b)** Average gains in accuracy across video streams
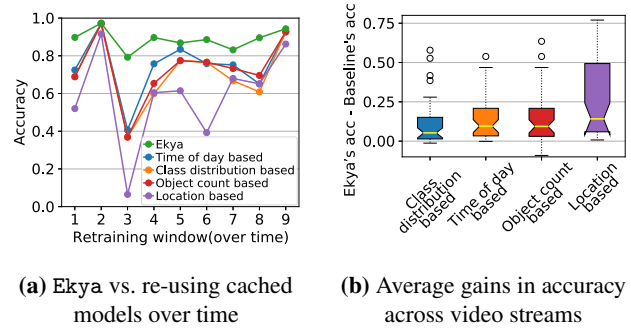
Figure 12: **Ekya vs. re-using cached models. Compared to cached-model selection techniques, models retrained with Ekya maintain a consistently high accuracy, since it fully leverages the latest training data and is thus more robust to data-drift.**

a 4G uplink (5.1 Mbps [65]) and downloading the trained ResNet18 models (398 Mb each [7]) over the 17.5 Mbps downlink [65] takes 432 seconds (even excluding the model retraining time), which already exceeds the retraining window.

To test on the Cityscapes dataset, we extend our simulator (§5) to account for network delays during retraining, and test with 8 videos and 4 GPUs. We use the conservative assumption that retraining in the cloud is "instantaneous" (cloud GPUs are powerful than edge GPUs). Table 3 lists the accuracies with cellular 4G links (both one and two subscriptions to meet the 400s retraining window) and a satellite link, which are both indicative of edge deployments [81].

For the cloud alternatives to match Ekya's accuracy, we will need to provision additional uplink capacity of 5×-10× and downlink capacity of 2×-4× (of the already expensive links). In summary, Ekya's edge-based solution is better than a cloud alternate for retraining in *both* accuracy and network usage (Ekya sends no data out of the edge), all while providing privacy for the videos. However, when the edge-cloud network has sufficient bandwidth, e.g., in an enterprise that is provisioned with a private leased connection, then using the cloud to retrain the models can be a viable design choice.

Ekya vs. Re-using pretrained models: An alternative to continuous retraining is to cache *pretrained* models and reuse them. We pre-train and cache a few tens of DNNs from earlier windows of the Waymo dataset and test four heuristics for selecting cached models. *Class-distribution*-based selection picks the cached DNN whose training data class distribution has the closest Euclidean distance with the current window's data. *Time-of-day*-based selection picks the cached

DNN whose training data time matches the current window. *Object-count*-based selection picks the cached DNN based on similar count of objects. *Location*-based selection picks the cached DNNs trained on the same city as the current window.

Figure 12a highlights the advantages of Ekya over different model selection schemes. We find that since time-of-day-based, object-count-based, and location-based model selection techniques are agnostic to the class distributions of training data of cached models, the selected cached models sometimes do not cover all classes in the current window. Even if we take class distribution into account when picking cached models, there are still substantial discrepancies in the appearances of objects between the current window and the history training data. For instance, object appearance can vary due to pose variations, occlusion or different lighting conditions. In Window 3 (Figure 12a), not only are certain classes underrepresented in the training data, but the lighting conditions are also adverse. Figure 12b presents a box plot of the accuracy difference between Ekya and model selection schemes, where the edges of the box represent the first and third quartiles, the waist is the median, the whiskers represent the maximum and minimum values and the circles represent any outliers. Ekya's continuous retraining of models is robust to scene specific data-drifts and achieves upto 26% higher mean accuracy.

7 Limitations and Discussion

Edge hierarchy with heterogeneous hardware. While Ekya's allocates GPU resources on a single edge, in practice, deployments typically consist of a *hierarchy* of edge devices [19]. For instance, 5G settings include an on-premise edge cluster, followed by edge compute at cellular towers, and then in the core network of the operator. The compute resources, hardware (e.g., GPUs, Intel VPUs [1], and CPUs) and network bandwidths change along the hierarchy. Thus, Ekya will have to be extended along two aspects: (*a*) multi-resource allocation to include both compute and the network in the edge hierarchy; and (*b*) heterogeneity in edge hardware.

Privacy of video data. As explained in §2.1, privacy of videos is important in real-world deployments, and Ekya's decision to retrain only on the edge device is well-suited to achieving privacy. However, when we extend Ekya to a hierarchy of edge clusters, care has to be taken to decide the portions of the retraining that can happen on edge devices that are *not* owned by the enterprise. Balancing the need for privacy with resource efficiency is a subject for future work.

Generality beyond vision workloads. Ekya's thief scheduler is generally applicable to DNN models since it only requires that the resource-accuracy function be strictly increasing wherein allocation of more resources to training results in increasing accuracy. This property holds true for *most* workloads (vision and language DNNs). However, when this property does *not* hold, further work is needed to prevent Ekya's microprofiler from making erroneous estimations and its thief scheduler from making sub-optimal allocations.

8 Related Work

1) ML training systems. For large scale scheduling of training in the cloud, model and data parallel frameworks [3, 10, 24, 50] and various resource schedulers [30, 31, 56, 69, 95, 97] have been developed. These systems, however, target different objectives than Ekya, like maximizing parallelism, fairness, or minimizing average job completion. Collaborative training systems [18, 51] work on decentralized data on mobile phones. They focus on coordinating the training between edge and the cloud, and not on training alongside inference.

2) Video processing systems. Prior work has built low-cost, high-accuracy and scalable video processing systems for the edge and cloud [22, 32, 37]. VideoStorm investigates quality-lag requirements in video queries [32]. NoScope exploits difference detectors and cascaded models to speedup queries [22]. Focus uses low-cost models to index videos [34]. Chameleon exploits correlations in camera content to amortize *profiling costs* [37]. Reducto [47] and DDS [25] seek to reduce edge-to-cloud traffic by intelligent frame sampling and video encoding. All of these works optimize only the inference accuracy or the system/network costs of DNN inference, unlike Ekya's focus on retraining. More recently, LiveNAS[41] deploys continuous retraining to update video upscaling models, but focuses on efficiently allocating client-server bandwidth to different subsamples of a single video stream. Instead, Ekya focuses on GPU allocation for maximizing retrained accuracy across multiple video streams.

3) Hyper-parameter optimization. Efficient exploration of hyper-parameters is crucial in training systems to find the model with the best accuracy. Techniques range from simple grid or random search [17], to more sophisticated approaches using random forests [35], Bayesian optimization [85, 88], probabilistic modelling [71], or non-stochastic infinite-armed bandits [46]. Unlike the focus of these techniques on finding the hyper-parameters with the highest accuracy, our focus is on resource allocation. Further, we are focused on the inference accuracy over the retrained window, where producing the best retrained model often turns out to be sub-optimal.

4) Continuous learning. Machine learning literature on continuous learning adapts models as new data comes in. A common approach used is transfer learning [33, 51, 72, 74]. Research has also been conducted on handling catastrophic forgetting [43, 79], using limited amount of training data [73, 89], and dealing with class imbalance [16, 92]. Ekya builds atop continuous learning techniques for its scheduling and implementation, to enable them in edge deployments.

9 Acknowledgements

References

[1] Azure percept. https://azure.microsoft.com/en-us/services/azure-percept/.

[2] Google ai blog: Custom on-device ml models with learn2compress. https://ai.googleblog.com/2018/05/custom-on-device-ml-models.html. (Accessed on 03/09/2021).

[3] MxNet: a flexible and efficient library for deep learning. https://mxnet.apache.org/.

[4] Nvidia multi-process service. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf. (Accessed on 09/16/2020).

[5] Reducing edge compute cost for live video analytics. https://techcommunity.microsoft.com/t5/internet-of-things/live-video-analytics-with-microsoft-rocket-for-reducing-edge-ba-p/1522305. (Accessed on 03/09/2021).

[6] scipy.optimize.nnls — scipy v1.5.2 reference guide. https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.nnls.html. (Accessed on 09/17/2020).

[7] torchvision.models — pytorch 1.6.0 documentation. https://pytorch.org/docs/stable/torchvision/models.html. (Accessed on 09/16/2020).

[8] A Comprehensive List of Hyperparameter Optimization & Tuning Solutions. https://medium.com/@mikkokotila/a-comprehensive-list-of-hyperparameter-optimization-tuning-solutions-88e067f19d9, 2018.

[9] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Fair allocation of multiple resource types. In *USENIX NSDI*, 2011.

[10] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *USENIX OSDI*, 2016.

[11] Achieving Compliant Data Residency and Security with Azure.

[12] AI and Compute. https://openai.com/blog/ai-and-compute/, 2018.

[13] G. Ananthanarayanan, V. Bahl, P. Bodík, K. Chintalapudi, M. Philipose, L. R. Sivalingam, and S. Sinha. Real-time Video Analytics – the killer app for edge computing. *IEEE Computer*, 2017.

[14] AWS Outposts. https://aws.amazon.com/outposts/.

[15] Azure Stack Edge. https://azure.microsoft.com/en-us/services/databox/edge/.

[16] E. Belouadah and A. Popescu. IL2M: Class Incremental Learning With Dual Memory. In *IEEE ICCV*, 2019.

[17] J. Bergstra and Y. Bengio. Random Search for Hyper-Parameter Optimization. *J. Mach. Learn. Res.*, 13:281–305, 2012.

[18] K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. Kiddon, J. Konecný, S. Mazzocchi, H. B. McMahan, T. V. Overveldt, D. Petrou, D. Ramage, and J. Roselander. Towards Federated Learning at Scale: System Design. In *SysML*, 2019.

[19] Chien-Chun Hung, Ganesh Ananthanarayanan, Peter Bodik, Leana Golubchik, Minlan Yu, Paramvir Bahl, Matthai Philipose. Videoedge: Processing camera streams using hierarchical clusters. In *ACM/IEEE SEC*, 2018.

[20] CLIFFORD, M. J., PERRONS, R. K., ALI, S. H.,ANDGRICE, T. A. Extracting Innovations: Mining, Energy, and Technological Changein the Digital Age. In *CRC Press*, 2018.

[21] cnn-benchmarks. https://github.com/jcjohnson/cnn-benchmarks#resnet-101, 2017.

[22] D. Kang, J. Emmons, F. Abuzaid, P. Bailis and M. Zaharia. Noscope: Optimizing neural network queries over video at scale. In *VLDB*, 2017.

[23] D Maltoni, V Lomonaco. Continuous learning in single-incremental-task scenarios. In *Neural Networks*, 2019.

[24] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. W. Senior, P. A. Tucker, K. Yang, and A. Y. Ng. Large Scale Distributed Deep Networks. In *NeurIPS*, 2012.

[25] K. Du, A. Pervaiz, X. Yuan, A. Chowdhery, Q. Zhang, H. Hoffmann, and J. Jiang. Server-driven video streaming for deep learning inference. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 557–570, 2020.

[26] Edge Computing at Chick-fil-A. https://medium.com/@cfatechblog/edge-computing-at-chick-fil-a-7d67242675e2. 2019.

[27] Ganesh Ananthanarayanan, Victor Bahl, Yuanchao Shu, Franz Loewenherz, Daniel Lai, Darcy Akers, Peiwei Cao, Fan Xia, Jiangbo Zhang, Ashley Song. Traffic Video Analytics – Case Study Report. 2019.

[28] GI Parisi, R Kemker, JL Part, C Kanan, S Wermter . Continual lifelong learning with neural networks: A review. In *Neural Networks*, 2019.

[29] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. Karro, and D. Sculley. Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, page 1487–1495, New York, NY, USA, 2017. Association for Computing Machinery.

[30] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. In *ACM SIGCOMM*, 2014.

[31] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. H. Liu, and C. Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *USENIX NSDI*, 2019.

[32] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodík, Matthai Philipose, Victor Bahl, Michael Freedman. Live video analytics at scale with approximation and delay-tolerance. In *USENIX NSDI*, 2017.

[33] G. Hinton, O. Vinyals, and J. Dean. Distilling the Knowledge in a Neural Network. In *NeurIPS Deep Learning and Representation Learning Workshop*, 2015.

[34] K. Hsieh, G. Ananthanarayanan, P. Bodík, S. Venkataraman, P. Bahl, M. Philipose, P. B. Gibbons, and O. Mutlu. Focus: Querying Large Video Datasets with Low Latency and Low Cost. In *USENIX OSDI*, 2018.

[35] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential Model-Based Optimization for General Algorithm Configuration. In *Learning and Intelligent Optimization*, 2011.

[36] Joseph Redmon, Ali Farhadi . Yolo9000: Better, faster, stronger. In *CVPR*, 2017.

[37] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodík, Siddhartha Sen, Ion Stoica. Chameleon: Scalable adaptation of video analytics. In *ACM SIGCOMM*, 2018.

[38] Junjue Wang, Ziqiang Feng, Shilpa George, Roger Iyengar, Pillai Padmanabhan, Mahadev Satyanarayanan. Towards scalable edge-native applications. In *ACM/IEEE Symposium on Edge Computing*, 2019.

[39] K He, X Zhang, S Ren, J Sun . Deep residual learning for image recognition. In *CVPR*, 2016.

[40] M. Khani, P. Hamadanian, A. Nasr-Esfahany, and M. Alizadeh. Real-time video inference on edge devices via adaptive model streaming. *arXiv preprint arXiv:2006.06628*, 2020.

[41] J. Kim, Y. Jung, H. Yeo, J. Ye, and D. Han. Neural-enhanced live streaming: Improving live video ingest via online learning. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 107–125, New York, NY, USA, 2020. Association for Computing Machinery.

[42] Konstantin Shmelkov, Cordelia Schmid, Karteek Alahari . Incremental learning of object detectors without catastrophic forgetting. In *ICCV*, 2017.

[43] J. Lee, J. Yoon, E. Yang, and S. J. Hwang. Lifelong Learning with Dynamically Expandable Networks. In *ICLR*, 2018.

[44] A. Li, O. Spyra, S. Perel, V. Dalibard, M. Jaderberg, C. Gu, D. Budden, T. Harley, and P. Gupta. A generalized framework for population based training. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '19, page 1791–1799, New York, NY, USA, 2019. Association for Computing Machinery.

[45] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *J. Mach. Learn. Res.*, 18(1):6765–6816, Jan. 2017.

[46] L. Li, K. G. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *J. Mach. Learn. Res.*, 18:185:1–185:52, 2017.

[47] Y. Li, A. Padmanabhan, P. Zhao, Y. Wang, G. H. Xu, and R. Netravali. Reducto: On-camera filtering for resource-efficient real-time video analytics. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 359–376, 2020.

[48] R. Liaw, R. Bhardwaj, L. Dunlap, Y. Zou, J. E. Gonzalez, I. Stoica, and A. Tumanov. Hypersched: Dynamic resource reallocation for model development on a deadline. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 61–73, New York, NY, USA, 2019. Association for Computing Machinery.

[49] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg. Ssd: Single shot multibox detector. In *European Conference on Computer Vision*, pages 21–37. Springer, 2016.

[50] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning in the cloud. *PVLDB*, 5(8):716–727, 2012.

[51] Y. Lu, Y. Shu, X. Tan, Y. Liu, M. Zhou, Q. Chen, and D. Pei. Collaborative learning between cloud and end devices: an empirical study on location prediction. In *ACM/IEEE SEC*, 2019.

[52] M McCloskey, NJ Cohen. Catastrophic interference in connectionist networks: The sequential learning problem. In *Psychology of learning and motivation*, 1989.

[53] M Sandler, A Howard, Menglong Zhu, Andrey Zhmoginov, Liang-Chieh Chen . Mobilenetv2: Inverted residuals and linear bottlenecks. In *CVPR*, 2018.

[54] M. J. Magazine and M. Chern. A note on approximation schemes for multidimensional knapsack problems. *Math. Oper. Res.*, 9(2), 1984.

[55] K. Mahajan, A. Balasubramanian, A. Singhvi, S. Venkataraman, A. Akella, A. Phanishayee, and S. Chawla. Themis: Fair and efficient GPU cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 289–304, Santa Clara, CA, Feb. 2020. USENIX Association.

[56] K. Mahajan, A. Singhvi, A. Balasubramanian, S. Venkataraman, A. Akella, A. Phanishayee, and S. Chawla. Themis: Fair and efficient GPU cluster scheduling for machine learning workloads. In *USENIX NSDI*, 2020.

[57] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele . The cityscapes dataset for semantic urban scene understanding. In *CVPR*, 2016.

[58] Measuring Fixed Broadband - Eighth Report, FEDERAL COMMUNICATIONS COMMISSION OFFICE OF ENGINEERING AND TECHNOLOGY. https://www.fcc.gov/reports-research/reports/measuring-broadband-america/measuring-fixed-broadband-eighth-report. 2018.

[59] Microsoft-Rocket-Video-Analytics-Platform. https://github.com/microsoft/Microsoft-Rocket-Video-Analytics-Platform.

[60] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, Quoc V. Le. Mnasnet: Platform-aware neural architecture search for mobile. In *CVPR*, 2019.

[61] Mingxing Tan, Quoc V. Le . Efficientnet: Rethinking model scaling for convolutional neural networks. In *ICML*, 2019.

[62] U. Misra, R. Liaw, L. Dunlap, R. Bhardwaj, K. Kandasamy, J. E. Gonzalez, I. Stoica, and A. Tumanov. *RubberBand: Cloud-Based Hyperparameter Tuning*, page 327–342. Association for Computing Machinery, New York, NY, USA, 2021.

[63] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica. Ray: A distributed framework for emerging ai applications. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 561–577, USA, 2018. USENIX Association.

[64] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun . Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *ECCV*, 2018.

[65] OPENSIGNAL. Mobile Network Experience Report . https://www.opensignal.com/reports/2019/01/usa/mobile-network-experience. 2019.

[66] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[67] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, Jan Kautz. Pruning convolutional neural networks for resource efficient inference. In *ICLR*, 2017.

[68] Pei Sun and Henrik Kretzschmar and Xerxes Dotiwalla and Aurelien Chouard and Vijaysai Patnaik and Paul Tsui and James Guo and Yin Zhou and Yuning Chai and Benjamin Caine and Vijay Vasudevan and Wei Han and Jiquan Ngiam and Hang Zhao and Aleksei Timofeev and Scott Ettinger and Maxim Krivokon and Amy Gao and Aditya Joshi and Yu Zhang and Jonathon Shlens and Zhifeng Chen and Dragomir Anguelov. Scalability in perception for autonomous driving: Waymo open dataset, 2019.

[69] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *ACM EuroSys*, 2018.

[70] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo. Optimus: An efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[71] J. Rasley, Y. He, F. Yan, O. Ruwase, and R. Fonseca. HyperDrive: exploring hyperparameters with POP scheduling. In *ACM/IFIP/USENIX Middleware*, 2017.

[72] Ravi Teja Mullapudi, Steven Chen, Keyi Zhang, Deva Ramanan, Kayvon Fatahalian. Online model distillation for efficient video inference. In *ICCV*, 2019.

[73] S. V. Ravuri and O. Vinyals. Classification accuracy score for conditional generative models. 2019.

[74] A. S. Razavian, H. Azizpour, J. Sullivan, and S. Carlsson. CNN features off-the-shelf: an astounding baseline for recognition. In *IEEE CVPR Workshop*, 2014.

[75] J. Redmon and A. Farhadi. Yolov3: An incremental improvement, 2018.

[76] Residential landline and fixed broadband services . https://www.ofcom.org.uk/__data/assets/pdf_file/0015/113640/landline-broadband.pdf. 2019.

[77] RM French. Catastrophic forgetting in connectionist networks. In *Trends in cognitive sciences*, 1999.

[78] H. Robbins. Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society*, 58(5), 1952.

[79] Ronald Kemker, Marc McClure, Angelina Abitino, Tyler L. Hayes, and Christopher Kanan. Measuring catastrophic forgetting in neural networks. In *AAAI*, 2018.

[80] H. F. Scheduler. https://hadoop.apache.org/docs/r2.4.1/hadoop-yarn/hadoop-yarn-site/FairScheduler.html.

[81] Shadi Noghabi, Landon Cox, Sharad Agarwal, Ganesh Ananthanarayanan. The emerging landscape of edge-computing. In *ACM SIGMOBILE GetMobile*, 2020.

[82] H. Shen, S. Han, M. Philipose, and A. Krishnamurthy. Fast video classification via adaptive cascading of deep models. In *CVPR*, 2017.

[83] Shivangi Srivastava, Maxim Berman, Matthew B. Blaschko, Devis Tuia . Adaptive compression-based lifelong learning. In *BMVC*, 2019.

[84] Si Young Jang, Yoonhyung Lee, Byoungheon Shin, Dongman Lee, Dionisio Vendrell Jacinto . Application-aware iot camera virtualization for video analytics edge computing. In *ACM/IEEE SEC*, 2018.

[85] J. Snoek, H. Larochelle, and R. P. Adams. Practical bayesian optimization of machine learning algorithms. In *NIPS*, 2012.

[86] Song Han, Huizi Mao, William J. Dally . Accelerating very deep convolutional networks for classification and detection. In *ICLR*, 2017.

[87] Sweden Data Collection & Processing.

[88] K. Swersky, R. Kiros, N. Satish, N. Sundaram, M. M. A. Patwary, and R. P. Adams. Scalable Bayesian Optimization Using Deep Neural Networks. In *ICML*, 2015.

[89] Sylvestre-Alvise Rebuffi, Alexander Kolesnikov, Georg Sperl, Christoph H. Lampert. icarl: Incremental classifier and representation learning. In *CVPR*, 2017.

[90] The Future of Computing is Distributed. https://www.datanami.com/2020/02/26/the-future-of-computing-is-distributed/, 2020.

[91] H. Wang, A. Kembhavi, A. Farhadi, A. L. Yuille, and M. Rastegari. Elastic: Improving cnns with dynamic scaling policies. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2258–2267, 2019.

[92] Y. Wu, Y. Chen, L. Wang, Y. Ye, Z. Liu, Y. Guo, and Y. Fu. Large scale incremental learning. In *IEEE CVPR*, 2019.

[93] Xi Yin, Xiang Yu, Kihyuk Sohn, Xiaoming Liu and Manmohan Chandraker. Feature transfer learning for face recognition with under-represented data. In *IEEE CVPR*, 2019.

[94] Xiangyu Zhang, Jianhua Zou, Kaiming He, and Jian Sun. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *IEEE PAMI*, 2016.

[95] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, F. Yang, and L. Zhou. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *USENIX OSDI*, 2018.

[96] Z. Li and D. Hoiem . Learning without forgetting. In *ECCV*, 2016.

[97] H. Zhang, L. Stafman, A. Or, and M. J. Freedman. SLAQ: quality-driven scheduling for distributed machine learning. In *SoCC*, 2017.

[98] X. Zhang, X. Zhou, M. Lin, and J. Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 6848–6856, 2018.

Notation	Description
$\mathcal{V}$	Set of video streams
v	A video stream ($v \in \mathcal{V}$)
T	A retraining window with duration $\|T\|$
Γ	Set of all retraining configurations
γ	A retraining configuration ($\gamma \in \Gamma$)
Λ	Set of all inference configurations
λ	An inference configuration ($\lambda \in \Lambda$)
$\mathcal{G}$	Total number of GPUs
δ	The unit for GPU resource allocation
$A_T(v, \gamma, \lambda, \mathcal{R}, I)$	Inference accuracy for video v for given configurations and allocations
$C_T(v, \gamma, \lambda)$	Compute cost in GPU-time for video v for given configurations and allocations
$\phi_{v\gamma\lambda\mathcal{R}I}$	A set of binary variables ($\phi_{v\gamma\lambda\mathcal{R}I} \in \{0, 1\}$). $\phi_{v\gamma\lambda\mathcal{R}I} = 1$ iff we use retraining config γ, inference config λ, $\mathcal{R}\delta$ GPUs for retraining, $I\delta$ GPUs for inference for video v

Table 4: Notations used in Ekya's description.

[99] Ziwei Liu, Zhongqi Miao, Xiaohang Zhan, Jiayun Wang, Boqing Gong, Stella X. Yu . Large-scale long-tailed recognition in an open world. In *CVPR*, 2019.

A Thief Scheduler

A.1 Complexity Analysis.

Assuming all the $A_T(v, \gamma, \lambda, \mathcal{R}, I)$ values are known, the above optimization problem can be reduced to a multi-dimensional binary knapsack problem, a NP-hard problem [54]. Specifically, the optimization problem is to pick binary options ($\phi_{v\gamma\lambda\mathcal{R}I}$) to maximize overall accuracy while satisfying two capacity constraints (the first and second constraints in Eq 1). In practice, however, getting all the $A_T(v, \gamma, \lambda, \mathcal{R}, I)$ is *infeasible* because this requires training the edge DNN using all retraining configurations and running inference using all the retrained DNNs with all possible GPU allocations and inference configurations.

The uncertainty of $A_T(v, \gamma, \lambda, \mathcal{R}, I)$ resembles the multi-armed bandits (MAB) problem [78] to maximize the expected rewards given a limited number of trials for a set of options. Our optimization problem is more challenging than MAB for two reasons. First, unlike the MAB problem, the cost of trials ($C_T(v, \gamma, \lambda)$) varies significantly, and the optimal solution may need to choose cheaper yet less rewarding options to maximize the overall accuracy. Second, getting the reward $A_T(v, \gamma, \lambda, \mathcal{R}, I)$ after each trial requires "ground truth" labels that are obtained using the large golden model, which can only be used judiciously on resource-scarce edges (§2.2).

In summary, our optimization problem is computationally more complex than two fundamentally challenging problems (multi-dimensional knapsack and multi-armed bandits).

Algorithm 2: PickConfigs

Data: Resource allocations in temp_alloc[], configurations (Γ and Λ), retraining window T, videos V

Result: Chosen configs $\forall v \in V$, average accuracy over T

1 chosen_accuracies[] ←{ }; chosen_configs[] ←{ };
2 **for** v *in* V[] **do**
3 $\quad$ infer_config_pool[] = Λ.**where**(resource_cost < temp_alloc[v.inference_job] && accuracy ≥ a_{MIN});
4 $\quad$ infer_config = **max**(infer_config_pool, **key**=accuracy);
5 $\quad$ best_accuracy = 0;
6 $\quad$ **for** train_config *in* Γ **do**
$\quad\quad$ /* Estimate accuracy of inference/training config pair over retraining window   */
7 $\quad\quad$ accuracy = EstimateAccuracy(train_config, infer_config, temp_alloc[v.training_job], T);
8 $\quad\quad$ **if** accuracy > best_accuracy **then**
9 $\quad\quad\quad$ best_accuracy = accuracy;
10 $\quad\quad\quad$ best_train_config = train_config;
11 $\quad$ chosen_accuracies[v] = best_accuracy;
12 $\quad$ chosen_configs[v] = {infer_config, best_train_config};
13 **return** chosen_configs[], **mean**(chosen_accuracies[]);

YuZu: Neural-Enhanced Volumetric Video Streaming

Anlan Zhang[1]   Chendong Wang[1]*   Bo Han[2]   Feng Qian[1]

[1]*University of Minnesota, Twin Cities*   [2]*George Mason University*

Abstract

Differing from traditional 2D videos, volumetric videos provide true 3D immersive viewing experiences and allow viewers to exercise six degree-of-freedom (6DoF) motion. However, streaming high-quality volumetric videos over the Internet is extremely bandwidth-consuming. In this paper, we propose to leverage 3D super resolution (SR) to drastically increase the visual quality of volumetric video streaming. To accomplish this goal, we conduct deep intra- and inter-frame optimizations for off-the-shelf 3D SR models, and achieve up to 542× speedup on SR inference without accuracy degradation. We also derive a first Quality of Experience (QoE) model for SR-enhanced volumetric video streaming, and validate it through extensive user studies involving 1,446 subjects, achieving a median QoE estimation error of 12.49%. We then integrate the above components, together with important features such as QoE-driven network/compute resource adaptation, into a holistic system called YuZu that performs line-rate (at 30+ FPS) adaptive SR for volumetric video streaming. Our evaluations show that YuZu can boost the QoE of volumetric video streaming by 37% to 178% compared to no SR, and outperform existing viewport-adaptive solutions by 101% to 175% on QoE.

1 Introduction

Volumetric video is an emerging type of multimedia content. Unlike traditional videos and 360° panoramic videos [28, 53] that are 2D, every frame in a volumetric video consists of a 3D scene represented by a point cloud or a polygon mesh. The 3D nature of volumetric video enables viewers to exercise six degree-of-freedom (6DoF) movement: a viewer can not only "look around" by changing the yaw, pitch, and roll of the viewing direction, but also "walk" in the video by changing the translational position in 3D space. This leads to a truly immersive viewing experience. As the key technology of realizing telepresence [49], volumetric video has registered numerous applications. They can be viewed in multiple ways: through VR/MR (virtual/mixed reality) headsets or directly on PCs (similar to how we play 3D games).

Despite the potentials, streaming volumetric videos over the Internet faces a key challenge of high bandwidth consumption. High-quality volumetric content requires hundreds of Mbps bandwidth [27, 71]. To improve the Quality of Experience

(QoE) under limited bandwidth, prior work has mostly focused on viewport-adaptive streaming (*i.e.,* mainly streaming content that will appear in the viewport) [27, 41, 50]. However, they are ineffective when the entire scene falls inside the viewport. They also require 6DoF motion prediction that is unlikely to be accurate for fast motion. Some other proposals explored remote rendering [26, 52] (*e.g.,* having an edge node transcode 3D scenes into regular 2D frames). However, they require not only 6DoF motion prediction, but also edge/cloud-side transcoding that is difficult to scale, as summarized in Table 1.

In this paper, we employ a different and orthogonal approach toward improving the QoE of volumetric video streaming through *3D super resolution* (3D SR). SR was initially designed for improving the visual quality of 2D images [21, 65]. Recently, researchers in the computer vision community developed SR models for point clouds [43, 61, 63, 70]. This inspires us to employ SR for volumetric video streaming, as each frame of a volumetric video is typically either a point cloud or a 3D mesh.[1] Although there have been recent successful attempts on applying SR to 2D video streaming [22, 39, 68], 3D-SR-enhanced volumetric video streaming is unique and challenging due to the following reasons.

• There is a fundamental difference between *pixel-based* 2D frames and volumetric frames consisting of *unstructured 3D points*, making processing volumetric videos (even without SR) vastly different from 2D videos.

• Due to its 3D nature, the computation overhead of 3D SR is very high. We apply off-the-shelf 3D SR models to volumetric videos [1], and find that the runtime performance of 3D SR is unacceptably poor – achieving only ~0.1 frames per second (FPS) on a PC with a powerful GPU. In contrast, 2D SR can achieve line-rate upsampling by simply downscaling the model [68], but we find that only doing model downscaling is far from being adequate for line-rate 3D SR (*i.e.,* at 30+ FPS).

• Given its recent debut, there lacks research on basic infrastructures such as tools and models supporting volumetric video streaming. For example, there is no QoE model for volumetric videos that can guide bitrate adaptation or critical SR parameter selection; the wide range of factors affecting the QoE make constructing such a model quite challenging.

• There are other practical challenges to overcome, such as a lack of color produced by today's 3D SR models.

To address the above challenges, we begin by developing to

---

* Current affiliation: University of Wisconsin, Madison.

[1]We focus on point-cloud-based volumetric videos in this work, but the key concepts of YuZu also apply to mesh-based volumetric videos.

Schemes	Refs	Advantages (⊕) and Disadvantages (⊖)
Direct Streaming	N/A	⊕ Easy to implement, best QoE (if bandwidth is sufficient). ⊖ Highest network bandwidth (BW) usage.
Direct + VA	[27, 41]	⊕ Lower BW usage. ⊖ BW saving depends on user's motion, QoE depends on motion prediction.
Direct + SR	YuZu	⊕ Good QoE, further lower BW usage, adaptively trades compute resource for BW. ⊖ Requires training.
Remote Rendering	[26, 52]	⊕ Lowest BW usage. ⊖ QoE depends on motion prediction, need edge support (poor scalability).

Table 1: Four categories of volumetric video streaming approaches (VA = Viewport Adaptation; SR = Super Resolution).

our knowledge a first QoE model for assessing SR-enhanced volumetric video streaming. The model takes into account a variety of factors that may affect the QoE, such as video resolution (*i.e.,* point density)[2], viewing distance, upsampling ratio, SR-incurred distortion, and QoE metrics from traditional video streaming. We validate our model by conducting two IRB-approved user studies involving 1,446 voluntary participants from 40 countries, using a major genre of volumetric content, *i.e.,* portraits of single/multiple people. The validation results confirm its accuracy, with a median QoE estimation error of 12.49%. Our user studies offer definitive evidence that 3D SR can significantly boost the QoE of volumetric video streaming.

Next, we design, implement, and evaluate YuZu, which is to our knowledge a first SR-enhanced volumetric video streaming system. At its core, YuZu deeply optimizes the end-to-end upsampling pipeline in three aspects: *intra-frame SR*, *inter-frame SR*, and *network-compute resource management*, whose synergy helps drastically improve the runtime performance of SR while retaining the inference accuracy.

For **intra-frame SR**, our approaches are not limited to generic optimizations for deep learning models such as modifying SR models' structures for fast-paced SR. More importantly, we consider the factors that are unique to 3D SR and its data representation: we design a mechanism that leverages the low-resolution content (*i.e.,* the input to the SR model, which is typically discarded after being fed into the model) to reduce the SR model complexity; we also trim the preprocessing and post-processing stages of 3D SR and tailor them to volumetric video streaming. Note that these optimizations are generic, applicable to all the 3D SR models we have investigated [43, 61, 63, 70].

For **inter-frame SR**, YuZu speeds up SR by caching and reusing 3D SR results across consecutive frames. Realizing that none of the 2D inter-frame encoding techniques can be directly applied to volumetric videos, we design an effective inter-frame content reference scheme for SR-enhanced point cloud streams, followed by robust criteria determining whether SR results can be reused between two frames. We then extend reusing SR results from two to multiple consecutive frames through a dynamic-programming-based optimization. The synergy of the above intra- and inter-frame acceleration schemes fills the huge gap between off-the-shelf 3D SR models' performance and what is required for line-rate upsampling of point cloud streams.

YuZu further performs **network-compute resource man-**

agement through making judicious decisions about the quality level of the to-be-fetched content and its upsampling ratio. These two decision dimensions are subject to the dynamic network bandwidth and limited compute resources, respectively, which need to be jointly considered given their complex tradeoffs – a unique challenge compared to traditional adaptive bitrate (ABR) video streaming. YuZu takes a QoE-driven approach by maximizing the utility function derived from our QoE model. To solve the underlying optimization problem in real time, we develop a hybrid, two-stage algorithm that employs coarse-grained and fine-grained search at different time to efficiently find a good approximate solution. In addition, YuZu performs fast colorization of SR results through efficient nearest point search.

We implement the above components and integrate them into YuZu in 10,848 lines of code. Our extensive evaluations indicate that YuZu can achieve line-rate, adaptive, high-quality 3D SR. We highlight key evaluation results as follows.
• Our user study suggests that 3D SR can boost the volumetric video QoE by 37% to 178% compared to no SR.
• Our optimizations speed up 3D SR by 140× to 542× and reduce GPU memory usage by 68% to 90% with no accuracy degradation, compared to the vanilla SR models [43, 61].
• Compared to a recently proposed viewport-adaptive volumetric video streaming system [27], YuZu improves the QoE by 100.6% to 174.9%.

To summarize, we make the following contributions.
• We build an empirical QoE model for SR-enhanced volumetric videos, and validate it through large-scale user studies involving 1,446 participants. We build our models using volumetric content of single/multiple human portraits, a major application of volumetric video streaming. Note that the model can be applied to non-SR volumetric videos belonging to the same genre, with an SR ratio of 1.
• We propose and design YuZu, an SR-enhanced, QoE-aware volumetric video streaming system.
• We implement YuZu, and conduct extensive evaluations for its QoE improvement and runtime performance.

2 Background and Motivation

Recently, the computer vision community extended SR to *static* point clouds [43, 61, 63, 70]. When applied to a video v, SR trains offline a deep neural network (DNN) model M that *upsamples* low-resolution frames $L(v)$ to high-resolution ones $H(v)$, using the original (high-resolution) frames $F(v)$ for training. In the online inference, the server sends M and $L(v)$ to the client, which infers $H(v) = M(L(v))$. SR leverages the overfitting property of DNN to ensure that $H(v)$ is highly

---

[2]The resolution of a point cloud is defined as its point density; the resolution of a volumetric video is the avg. resolution of its point cloud frames.

similar to $F(v)$. It achieves bandwidth reduction (or QoE improvement when bandwidth remains the same) since the combined size of M and $L(v)$ is much smaller than $F(v)$.

We start with a straightforward approach: applying PU-GAN [43], a state-of-the-art 3D SR model, to upsample every point cloud frame of a volumetric video. PU-GAN operates by dividing the entire point cloud of a frame into smaller *patches*, each consisting of a subset of points. Both SR training and inference are performed on a per-patch (as opposed to a per-frame) basis, *i.e.,* each patch is upsampled individually. Its DNN model is based on a generative adversarial network (GAN) and realizes three key stages: feature extraction, feature expansion, and point set generation.

We next describe a case study using PU-GAN to motivate YuZu. Our testing video was captured by three depth cameras. It has 3,622 frames, each consisting of ~100K points depicting a performing actor. We use all its frames to train a PU-GAN model. We set the SR ratio (*i.e.,* upsampling ratio) to 4, making the input and output point clouds consist of roughly 25K and 100K points, respectively. We have both positive and negative findings from this case study. On the positive side, the model can accurately reconstruct each individual frame, *i.e.,* each upsampled point cloud is highly similar to the original one in terms of the geometric structure, as quantified by the Earth Mover's Distance (EMD [54]):

$$\mathcal{L}_{EMD}(I,G) = \min_{\phi:I \to G} \frac{1}{|I|} \sum_{x \in I} \|x - \phi(x)\|_2 \quad (1)$$

where I and G are the upsampled point cloud and the ground truth, respectively; $\phi : I \to G$ is a bijection from the points in I to those in G. The average EMD value across all frames is 1.47cm, which confirms good upsampling accuracy [43]; it is also verified by our IRB-approved user studies (§4.2). Also encouragingly, we find that SR indeed achieves significant bandwidth savings. For this 2-minute video, the compressed sizes of $F(v)$, M, and $L(v)$ are 1.40 GB, 560 KB, and 0.36 GB, respectively, leading to a bandwidth reduction of 74.2%.

Despite the above encouraging results, we notice three major issues from the above case study.
• **A Lack of Quality-of-Experience (QoE) Model.** For traditional 2D video streaming, there exist numerous studies on modeling the viewer's QoE [15, 18, 69]. In contrast, volumetric videos are still in their infancy. There is a lack of generic QoE models that researchers can leverage, not to mention a lack of understanding of how SR impacts QoE.
• **Unacceptably Poor Runtime Performance.** 3D SR models are computationally much more heavyweight than 2D SR models. When applying PU-GAN to the above video, the runtime performance is extremely poor. On a machine with an NVIDIA 2080Ti GPU, the upsampling FPS is only 0.1, far below the desired FPS of at least 30. Besides, the GPU memory usage of PU-GAN is 7GB (out of the 11GB available memory of 2080Ti). This is one reason why all the off-the-shelf 3D SR models operate on a per-patch basis, as this saves memory compared to processing a full frame.

• **No Color Support.** We find that no existing 3D SR model can restore the color information of upsampled point cloud.

Note that the last two limitations are common in that they also apply to all other 3D SR models for point clouds that we have examined, such as MPU [61] and PU-Net [70].

3 YuZu **Overview**

YuZu is to our knowledge the first SR-enhanced volumetric video streaming system. It streams video-on-demand volumetric content stored on an Internet server to client hosts. On the server side, the volumetric video is divided into *chunks* each consisting of a fixed number of frames (*i.e.,* point clouds encoded by schemes such as Octree [34,46] and k-d tree [35,44]). Each chunk is encoded into multiple versions with different resolutions (*i.e.,* point densities). The SR model training and volumetric content preprocessing (*e.g.,* patch reuse computation, see §5.2) are performed offline on the server side. Similar to a typical DASH server, the YuZu server is stateless (and thus scalable), and all the streaming logic runs on the client side. As shown in Figure 1, the client fetches from the server the video chunks, which can possibly be at a low resolution. Since 3D SR models typically operate on a per-patch basis, the client segments each frame into patches, upsamples them through 3D SR, efficiently colors them (§5.4), and renders them to the viewer.

To achieve line rate SR, YuZu employs novel optimizations tailored to SR-enhanced volumetric video streaming. Regarding *intra-frame optimizations*, off-the-shelf 3D SR models are strategically adapted; low-resolution patches before SR are properly leveraged instead of being discarded; and the patch generation is accelerated (§5.1). For *inter-frame optimizations*, previous SR results are judiciously reused (§5.2).

A crucial decision that YuZu must make is to determine what resolution (quality level) to fetch for each chunk, as well as which SR ratio to apply for upsampling each patch, subject to the resource constraints jointly imposed by the network and computation. YuZu addresses this through a principled, efficient, and QoE-driven discrete optimization framework (§5.3). The framework utilizes a first-of-its-kind QoE model that we derive from ratings of 1,446 real users (§4).

4 QoE Model for Volumetric Videos

For SR-enhanced volumetric video streaming, its QoE is affected by a wide range of factors. The large space formed by these factors and their interplay make constructing QoE models much more challenging than conventional videos.

4.1 An Empirical QoE Model

We first enumerate factors that may affect the QoE for SR-enhanced volumetric video streaming. They are derived based on the domain knowledge of SR and our communication with other volumetric video viewers.
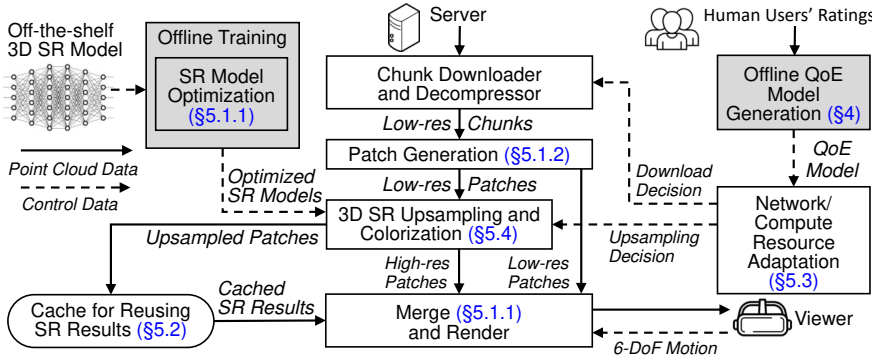
Figure 1: The system architecture of YuZu.

Age	18-25: 21.8%, 26-30: 29.0%, 31-35: 20.4%, 35+: 28.8%
Gender	Male: 60.3%, Female: 39.2%, Other: 0.5%
Country (40 Total)	US: 55.0%, IN: 28.1%, BR: 5.0%, IT: 2.7%, UK: 1.2%, DE: 1.0%, CA: 0.9%, Other: 6.1%
Education	Bachelor: 59.1%, Master: 23.8%, Other: 17.1%

Table 2: Demographics of the 1,446 subjects in our user studies.

- **Point Density**. Similar to 2D image resolution, a 3D object with a higher point density (resolution) contains more details and thus offers a better QoE.

- **Viewing Distance.** As the viewing distance increases, a rendered 3D object becomes smaller in the displayed view, and is thus less sensitive to quality degradation.

- **SR Ratio and Distortion.** A higher SR ratio leads to a higher point density (and thus more QoE gain), but also potentially higher distortions (and thus more QoE loss).

- **Artifacts caused by Patches.** As described in §2, a typical 3D SR model operates by upsampling individual point subsets called patches. If patches within a frame have non-uniform qualities (caused by different SR ratios), the perceived QoE will be affected.

- **Invisibility due to Finite Viewport and Occlusion.** Due to the 3D nature of volumetric videos, a viewer can see only content that is inside the viewport and not occluded. Outside-viewport or occluded content brings no impact on the QoE.

- **QoE Metrics for Regular Video Streaming.** They include factors such as stall and inter-frame quality switches [69].

Next, we develop an empirical QoE model that considers the above factors. Since SR is performed on a per-patch basis, we first model the QoE for each individual patch as:

$$q_{i,j} = g(d_{i,j}, r_{i,j}, \delta_{i,j}) - h(EMD, \delta_{i,j}) \quad (2)$$

where $q_{i,j}$ is the quality of patch j in frame i; $d_{i,j}$ is the patch's original point density before SR; $\delta_{i,j}$ is the viewing distance to the patch; $r_{i,j}$ is the SR ratio of the patch. Eq. 2 has two terms: $g(\cdot)$ considers the patch's perceived density after SR, and $h(\cdot)$ accounts for the QoE penalty incurred by SR distortion, quantified by the viewing distance and the EMD (Eq. 1) between the upsampled patch and the high-quality patch (ground truth). We empirically define $g(\cdot)$ and $h(\cdot)$ as:

$$g(d_{i,j}, r_{i,j}, \delta_{i,j}) = w_1(\delta_{i,j}) \times d_{i,j} \times r_{i,j} \quad (3)$$

$$h(EMD, \delta_{i,j}) = w_2(\delta_{i,j}) \times EMD \quad (4)$$

where $w_1(\delta_{i,j})$ and $w_2(\delta_{i,j})$ are weights parameterized on $\delta_{i,j}$. Intuitively, in Eq. 3, after SR, the perceived point density improves by a factor of $r_{i,j}$; the QoE gain brought by a higher point density after SR (Eq. 3) and the QoE penalty caused by SR distortion (Eq. 4) depend on the viewing distance.

Now given a single frame i, we define its quality Q_i as the average of all its visible patches' quality values:

$$Q_i = \frac{\sum_j v_{i,j} q_{i,j}}{\sum_j v_{i,j}} \quad (5)$$

where $v_{i,j} \in \{0, 1\}$ is 1 iff the patch is visible, *i.e.,* it falls inside the viewport and is not occluded by other patches. To account for the artifacts caused by patches, we define *inter-patch quality switch* I_i^{patch} as the quality variation across the visible patches within frame i. To account for inter-frame quality switches, we define *inter-frame quality switch* I_i^{frame} as the quality change from frame $i-1$ to frame i:

$$I_i^{patch} = \textbf{StdDev}\left(\{q_{i,j} | \forall j, v_{i,j} > 0\}\right) \quad (6)$$

$$I_i^{frame} = \|Q_i - Q_{i-1}\| \quad (7)$$

For a volumetric video playback, a possible way to model its overall QoE is a linear combination of Q_i, I_i^{patch}, I_i^{frame}, and I_i^{stall} (the stall of frame i). We choose a linear form that is widely used in 2D Internet videos [69]. Thus, we have

$$QoE = \sum_i Q_i - \sum_i \mu_p(\delta_i) I_i^{patch} - \sum_i \mu_f(\delta_i) I_i^{frame} - \sum_i \mu_s(\delta_i) I_i^{stall}$$
$$(8)$$

Note that depending on the viewing distance, the weights μ_p, μ_f, and μ_s may differ (*e.g.,* viewers may be more sensitive to stalls when watching a scene at a closer distance), so we parameterize the weights with the viewing distance. In Eq. 8, δ_i summarizes the viewing distances to all the patches in frame i. We empirically choose $\delta_i = (\sum_j v_{i,j} \delta_{i,j})/(\sum_j v_{i,j})$. Also note that the above model is generic and applicable to non-SR-enhanced and non-patch-based volumetric videos as it encompasses special cases without using SR ($r_{i,j}$=1) or patches (I_i^{patch}=0).

4.2 Model Validation through User Studies

We next conduct user studies with two purposes: validating our QoE model and deriving the model parameters. Our QoE model considers many factors as described in §4.1. The high-level approach of the user study is to let participants subjectively rate the QoE for all the combinations of the above factors' different degrees of impairments, and then use the

Scheme	1×1	1×2	1×3	1×4	2×1	2×2	3×1	4×1
Pt. density	25%	25%	25%	25%	50%	50%	75%	100%
SR ratio	-	×2	×3	×4	-	×2	-	-

Table 3: 8 impaired versions (except 4×1) of a video segment. In scheme $m \times n$, m is the point density level and n is SR ratio.

Videos: {*Long Dress*, *Loot* [1]; *Band*, *Haggle* [36]}
Avg. frame quality Q_i: 7 values uniformly selected from Table 3
Avg. distance $dist_{i,j}$: {1m, 2m, 3m, 4m}
Avg. inter-patch switch I_i^{patch}: {0.00, 0.45, 0.90}
Avg. inter-frame switch I_i^{frame}: {0.00, 0.45, 0.90}
Avg. stall I_i^{stall}: {0.00, 0.01, 0.03}

Table 4: The factors and their values selected for model validation.

subjects' ratings to train/validate our QoE model. We obtained IRB approvals for our studies. Instead of performing in-person studies, we conduct both studies online by letting users watch pre-generated videos capturing the rendered viewports (with impairments). We take this approach because: (1) it allows vastly scaling up the study, (2) it helps get diverse users worldwide, and (3) the IRB forbids in-person user studies during COVID-19. We have collected responses from 1,446 subjects, whose demographics are shown in Table 2.

We start by studying the QoE gain brought by SR. We have collected 512 subjects' responses with a total number of 57,344 ratings. The key finding is that SR can effectively boost the QoE. For example, at 1m, compared to 1×1, the (user-rated) QoE increases by 37%, 75%, 150% for 1×2, 1×3, and 1×4, respectively; 2×2 improves the QoE by 178% compared to 2×1. The details can be found in Appendix A.

Next, we validate the overall QoE model (Eq. 8). We choose four videos: *Long Dress* showing a dancing female, *Loot* showing a speaking male, *Band* showing three people playing instruments, and *Haggle* showing three people debating. *Long Dress* and *Loot* are obtained from the 8i dataset [1], each consisting of 800K points per frame for 10 seconds. *Band* and *Haggle* are from the CMU Panoptic dataset [36], each consisting of 300K and 100K points per frame, respectively; we select 10-second segments for our study. For each video, we create 8 versions listed in Table 3. Note that since the participants need to watch a large number of impaired copies, the video length (10 seconds) has to be short. Also note that the videos have different point densities, as we want to make the QoE model generic, applicable to different resolutions. We will experimentally verify this shortly. We use our optimized PU-GAN algorithm (details in §5.1) to perform upsampling and create video clips at 4K resolution for four viewing distances: 1m, 2m, 3m, and 4m, which are determined from a separate IRB-approved user study whose details are described in Appendix B. To maintain a fixed viewing distance d, we display the viewport at d meters in front of and facing the viewer. We design a survey using Qualtrics [11] and publish it on Amazon Mechanical Turk (AMT) [2].

We study the impact of all the factors in Eq. 8 on the QoE. Table 4 lists them and their impairment levels. They lead to a total of 756 combinations for each video segment. Since

letting subjects perform $\binom{756}{2}$ pairwise comparisons is infeasible, for each combination, we generate one video clip by putting the impaired version and the high-quality "ground truth" version (4×1, $I_i^{patch} = I_i^{frame} = I_i^{stall} = 0$, same viewing distance) side by side, in a random order. To generate the impaired version, we randomly add perturbations to the patches' quality levels to match the corresponding I_i^{patch} and I_i^{frame} values, and randomly inject stalls to match I_i^{stall}. We then ask each subject to watch 100 randomly selected video clips from the 756 clips of a randomly selected video segment. After watching each clip, the subject is asked to rate which side provides a better QoE through 7 choices ("left looks {much better, better, slightly better, similar to, slightly worse, worse, much worse} than right") If the impaired version is {similar to, slightly worse, worse, much worse} than the ground truth, we give the impaired version a score of {3,2,1,0}, respectively.

We have collected 934 subjects' responses with a total number of 93,400 ratings for the above survey published on AMT. For each viewing distance, we use the subjects' ratings to calculate the average score of each of the 756 impaired clips on a scale from 0 to 3, and use it as the QoE ground truth. We then perform 10-fold cross-validation to validate our QoE model (Eq. 8, trained using multi-variable linear regression) for each viewing distance. Figure 2 plots the CDF of the QoE prediction errors at each viewing distance. The median prediction error for 1m, 2m, 3m, 4m is 11.4%, 12.2%, 12.8%, and 12.9%, respectively. The (Person, Spearman) correlation coefficients between the ground-truth QoE score and the predicted QoE score are also high: (0.89, 0.89) at 1m, (0.87, 0.88) at 2m, (0.87, 0.88) at 3m, and (0.85, 0.85) at 4m.

The above QoE models are trained from all four videos. Table 5 shows the Spearman correlation coefficients between the ground-truth QoE and *cross-video* prediction results. We use the data of three videos to train a QoE model and use it to predict the QoE for the remaining video. The results indicate that the same QoE model and its parameters are applicable to volumetric content of the same genre (portraits of people – a major application of volumetric streaming – in our case). We also confirm that most parameters trained from different video segments are indeed quite similar, in spite of the segments' different point densities. When applied to other genres, the model's parameters may differ, as to be explored in our future work (the same happens to 2D videos [68]). Table 6 lists our final model's parameters trained using the entire dataset. The model will be used by YuZu.

5 System Design of YuZu

We now detail the system design of YuZu (Figure 1) that addresses the challenges we identified in §2.

5.1 Accelerating SR Upsampling

To accelerate 3D upsampling, we take a principled approach by exploring three orthogonal directions:
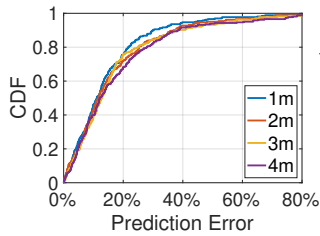
Figure 2: QoE prediction error using our model.

δ	D: Long Dress; L: Loot; B:Band; H:Haggle			
	$DBH \Rightarrow L$	$LBH \Rightarrow D$	$DLB \Rightarrow H$	$DLH \Rightarrow B$
1m	0.80	0.74	0.86	0.85
2m	0.76	0.71	0.87	0.87
3m	0.80	0.73	0.83	0.87
4m	0.78	0.71	0.76	0.80

Table 5: Spearman correlation coefficient between QoE ground truth and cross-video prediction. $XYZ \Rightarrow W$ means using the model trained from videos X, Y, and Z to predict video W's QoE.

δ	Long Dress + Loot + Band + Haggle				
	w_1	w_2	μ_p	μ_f	μ_s
1m	0.55	27.80	0.52	0.40	170.5
2m	0.42	39.83	1.05	0.91	149.8
3m	0.27	26.63	1.23	1.04	176.7
4m	0.16	17.17	0.47	0.06	304.1

Table 6: Parameters of the final model used in YuZu.

Figure 3: Using a 3× SR model to realize 4× SR.

- **Model Optimization.** How to simplify the upsampling logic while retaining the inference accuracy? (§5.1.1)
- **Data Reduction.** How to strategically feed less data to SR models with negligible impact on QoE? (§5.2)
- **Pre-processing and Post-processing Trimming.** How to simplify the sophisticated pre- and post-processing stages without incurring side effects on inferences? (§5.1.2)

Our optimizations can apply to all 3D SR models we have investigated [43, 61, 63, 70] and they are video-agnostic. In §7, we demonstrate the optimization results for two SR models: PU-GAN [43] and MPU [61].

5.1.1 SR Model Optimization

We take a "top-down" approach by first optimizing the model as a whole and then fine-tuning its detailed structure. For most machine learning models (including 2D SR), after performing an inference, the input is no longer needed and will be discarded. Our investigated 3D SR models [43, 61, 63, 70] make no exception. We instead make a fundamental observation regarding 3D point clouds. Different from a 2D image, a point cloud is a set of unstructured points, which means that point clouds can be *merged via a simple set union operation*. We also note that 3D SR's output points *refine and differ from* the input. Based on this key insight, we propose a simple yet effective optimization: YuZu merges the input low-density point cloud with the SR output in order to improve the visual quality, or to reduce the computation overhead while maintaining the same upsampling ratio. For example, as shown in Figure 3, to achieve 4× upsampling, instead of using a 4× SR model, we can use a (computationally more efficient) 3× SR model and merge the input with the output. Since SR exploits the overfitting nature of DNN, the spatial distributions of upsampled points and the ground truth are expected to be highly similar. By leveraging the input data and downgrading the SR ratio from 4× to 3×, we can achieve an acceleration of up to ~35% without hurting the SR accuracy (Figure 6). Note that in offline training, the loss function is computed *after* merging the input low-density point cloud with the SR output. This makes the trained models aware of and adaptive

to the merging process, improving the upsampling accuracy compared to computing the loss function before that.

Next, we explore modifying 3D SR model's DNN structure for inference acceleration. By profiling the inference time of PU-GAN, we find that its three stages, feature extraction, feature expansion, and point set generation, take 78.3%, 19.3%, and 2.4% of execution time, respectively (4× SR). Within the feature extraction stage that dominates the runtime overhead, most operations are *convolutions*. We make the same observation for other 3D SR models that we investigated [61, 63, 70].

To accelerate convolutions, we replace the original feature extraction, which (*e.g.,* in the case of PU-GAN) enhances the solution in PointNet++ [51] through dynamic graph convolution [56], with a recent proposal called spherical kernel function (SKF) [42]. SKF partitions a 3D space into multiple volumetric bins and specifies a learnable parameter to convolve the points in each bin. In contrast to continuous filter approaches (*e.g.,* multilayer perceptron) used in existing SR models, SKF is a *discrete* metric-based spherical convolutional kernel, and is thus computationally attractive for dense point clouds. Moreover, it is applicable to all the 3D SR models we examined. We find that SKF brings no degradation to the upsampling accuracy (§7.3). One reason may be that the kernel asymmetry of SKF facilitates learning fine geometric details of point clouds [42].

In addition to utilizing SKF, we conduct layer-by-layer profiling [22, 66] to fine-tune the SR model's performance-accuracy tradeoff. Take PU-GAN as an example. We remove the last two dense layers of feature extraction and several heavyweight convolution layers in the feature expansion stage, as they make limited contributions to the upsampling accuracy. We also judiciously remove a small number of expanded features to reduce the GPU memory footprint. For other 3D SR models, their model tuning follows a similar approach.

5.1.2 Trimming Pre- and Post-Processing

Recall from §2 that to ensure a manageable model complexity, a 3D SR model divides a point cloud into small patches as basic units for upsampling. We discover that as an important pre-processing step, the patch generation process incurs a high overhead. For example, PU-GAN generates the patches by applying kNN to the seeds created by downsampling. Since the generated patches may overlap, after upsampling, PU-

GAN needs to perform post-processing: it applies the furthest point sampling [48] to remove duplicated points.

To mitigate the above overhead, YuZu adopts a simple patch generation method. It divides the space into cubic cells, and assigns each non-empty cell (*i.e.,* a cell that contains points) to a patch. Compared to the default patch generation approaches used by PU-GAN and other 3D SR frameworks [43, 61], our approach runs very fast; it also brings no overlap among patches, thus eliminating the post-processing step (*i.e.,* overlap removal). In addition, the patches now have a simple geometry shape, so that they can be indexed, searched, and manipulated at runtime. Meanwhile, We find that our patch generation approach does not sacrifice the up-sampling accuracy and may even improve the accuracy compared to vanilla PU-GAN and MPU (§7.3). This is likely because cubic cells provide a more consistent structure for the patches, making it easier to perform SR. We also investigate several other patch generation methods based on Voronoi diagram [24] and 3D SIFT [55], but none outperforms our cubic-cell-based approach from either the performance or the accuracy perspective.

5.2 Caching and Reusing SR Results

Videos usually exhibit similarities across frames. We find that volumetric videos make no exceptions. This indicates rich opportunities for caching and reusing SR results.

At a high level, YuZu reuses SR results based on the similarity between patches, which is the basis of inter-frame encoding. Inter-frame similarity has been extensively studied and exploited in 2D videos. However, none of the 2D inter-frame encoding techniques can be directly applied to volumetric videos due to the fundamental difference between pixel-based 2D frames and volumetric frames consisting of unstructured points. There are very few studies on 3D inter-frame encoding [37, 46]; they are incompatible with YuZu's patch-based upsampling, and incur high complexity hindering line-rate decoding. Due to the above reasons, we design our own SR caching/reusing algorithm. Our algorithm is agnostic of and orthogonal to a specific SR model.

YuZu reuses 3D SR results on a per-patch basis to match the patch-based upsampling procedure. Recall from §5.1.2 that YuZu generates patches using 3D cubic cells. Let $p(i, j)$ denote patch j of frame i, and let $N(i, j)$ denote the number of points in $p(i, j)$. YuZu allows reusing the SR result of $p(i, j)$ for subsequent *consecutive patches at the same location*, *i.e.,* $p(i+1, j), p(i+2, j)$, and so on. YuZu restricts reusing patches only at the same location due to two considerations. First, we empirically observe that most patch similarities indeed occur at the same cell location; this makes the benefits (in terms of reduced SR overhead) of reusing a patch belonging to a different cell marginal. Second, allowing reusing a patch at a different cell will drastically increase the overhead of pre-computing the caching/reusing decisions.

We now describe YuZu's SR reuse algorithm. YuZu first determines offline the similarity of two patches. For each patch pair $(p(i, j), p(i+1, j))$, YuZu computes a Weighted Complete Bipartite Graph [17] $B : p(i, j) \rightarrow p(i+1, j)$, which we find to be suitable for dealing with unstructured points. In the bipartite graph, there is a directed edge from every point in $p(i, j)$ to every point in $p(i+1, j)$, and the weight of the edge is their Euclidean distance. We then calculate the minimum-weight matching (MWM) [57] for the graph, *i.e.,* finding $N(i, j)$ edges such that (1) these edges share no common vertices (points), and (2) the sum of their weights is minimized. Intuitively, the MWM identifies a transformation from $p(i, j)$ to $p(i+1, j)$ with a minimum moving distance for the points. The Hungarian algorithm [17] that computes the MWM has a complexity of $O(N^4)$ where $N = \max\{N(i, j), N(i+1, j)\}$. We instead employ a faster $O(N^2)$ approximation algorithm that is found to work well in practice.[3]

We call every edge in the MWM a point motion vector (PMV). A PMV differs from a 2D video's motion vector, which represents a macroblock in a frame based on the position of the same or a similar macroblock in another reference frame. Leveraging the PMVs, we determine that $p(i+1, j)$ and $p(i, j)$ are *similar* if three criteria are satisfied. (1) $N(i, j)$ and $N(i+1, j)$ differ by no more than $\eta_n\%$; (2) the average length of all the PMVs is smaller than η_a; (3) the top 90-percentile of the shortest PMV is smaller than η_v. These three criteria dictate that $p(i, j)$ and $p(i+1, j)$ have a similar number of points, and the points' collective motions are small. Figure 4 shows how η_a impacts EMD and the patch reuse ratio (% of patches that can reuse a previous SR result). As shown, increasing η_a increases the reuse ratio, but meanwhile decreases the accuracy. According to Figure 4, we set η_a to 0.01m to balance the performance and accuracy. Using similar methods, we empirically set η_n=10 and η_v=0.01m.

Next, we consider how to reuse an SR result across multiple patches belonging to consecutive frames. We define $sim_j(i_1, i_2) \in \{0, 1\}$ to be 1 if and only if $p(i_1, j)$ and $p(i_2, j)$ are similar, *i.e.,* satisfying the above three criteria where $i_2 > i_1$. Figure 5 shows an example of 6 consecutive patches at location j where $\forall 1 \le x < y \le 6 : sim_j(x, y) = 0$ except that $sim_j(1, 2), sim_j(2, 3), sim_j(2, 4)$, and $sim_j(2, 6)$ are 1. YuZu allows a patch's SR result to be reused across *consecutive* patches if they are all similar to the first patch. For example, Patches 3 and 4 can reuse Patch 2's SR result. However, YuZu does not let Patch 6 reuse Patch 2 because $sim_j(2, 5) = 0$. We make this design decision for two reasons. First, we observe that non-consecutive patches are unlikely to be similar in real volumetric videos. Second, supporting non-consecutive reuse requires computing $sim_j(x, y) \forall x < y$, making offline video processing slow.

We develop an algorithm that *minimizes the number of*

---

[3]The approximation algorithm sorts all the edges by their weights in ascending order. It then adds the edges to the MWM in that order and skips edges that share points with an existing edge in the matching, until every point in $p(i, j)$ or every point in $p(i+1, j)$ is in the MWM.
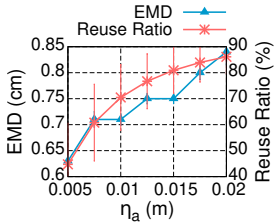
Figure 4: Impact of η_a (using the video in §2, 1×4 SR).
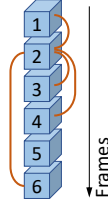
Figure 5: Reusing SR results across consecutive frames.

patches to be upsampled, to boost the online SR performance. For example, in Figure 5, the minimum number of patches to be upsampled is 4: Patches 1, 2, 5, and 6. YuZu efficiently and optimally solves this through dynamic programming (DP). Given n patches $p(1,j),...,p(n,j)$ and their sim_j information, let $u(i,j)$ be the minimum number of patches that need to be upsampled in $\{p(i,j),...,p(n,j)\}$ if we decide to upsample $p(i,j)$. Then $u(i,j)$ can be derived through DP as:

$$u(i,j) = \min\left\{u(i+1,j), \min_{i<k\leq n:\forall i<t\leq k:sim_j(i,t)=1}\{u(k+1,j)\}\right\} + 1$$
(9)

The RHS of Eq. 9 examines each patch following $p(i,j)$ and updates $u(i,j)$ if stopping reusing $p(i,j)$ at $p(k+1,j)$ yields a better $u(i,j)$. The search continues until hitting a patch that is not similar to $p(i,j)$. Eq. 9 can be solved backwards starting from $u(n+1,j)=0$. The solution is $u(1,j)$.

Since YuZu streams VoD volumetric content, all the above logic (calculating MWM, sim_j, and DP) is performed *offline* for each patch location j. Thus, there is no runtime overhead. The SR reuse decisions are sent to the client as meta data, which is only 0.5KB per frame for our testing video in §2.

5.3 Network/Compute Resource Adaptation

YuZu adapts to not only the fluctuating network condition (similar to the job of traditional bitrate adaptation algorithms [45, 64, 69]), but also the available compute resource, due to the high computation overhead of 3D SR. More importantly, these two dimensions incur a tradeoff: given a fixed playback deadline, should YuZu download high-resolution content, or download lower-resolution content and spend time upsampling it? Fortunately, our QoE model (§4.1) dictates how to quantitatively balance this tradeoff.

We first formulate an online network/compute adaptation problem. The video is divided into n chunks each consisting of f frames. To achieve fine-grained adaptation, each chunk is further spatially segmented into b *blocks* (*e.g.*, $b=5^3$), which are the atomic scheduling units in YuZu's adaptation algorithm. Each block consists of multiple patches (recall from §5.1.2 that each patch occupies a cubic cell). At runtime, YuZu considers all the blocks belonging to a finite horizon of the next w chunks, and searches for their quality and SR ratio assignments that maximize the QoE defined in Eq. 8. This formulation extends the model predictive control (MPC) scheme [69] that proves to be effective for traditional 2D

video streaming. The solution space is $O(8^{wb})$ (the 8 possible assignments are listed in Table 3).

We consider how to efficiently solve the above discrete optimization problem. An exhaustive search is clearly infeasible. Due to the large solution space, even the memorization approach (FastMPC [69]) is not practical. Another possibility is a learning-based approach such as Pensieve [45]. However, it requires offline training and may incur a non-trivial inference overhead. Moreover, a recent work [64] indicates that reinforcement learning based bitrate adaptation solutions do not necessarily outperform simple buffer-based approaches [33].

To overcome the above challenges, we develop a lightweight approximation algorithm. It executes in two stages: first determine the quality and SR ratios of to-be-downloaded chunks, and then fine-tune the SR ratios before upsampling. Specifically, in the first stage, *before downloading each chunk*, YuZu performs a *coarse-grained search* by assuming that all the blocks in each chunk have the same quality/SR-ratio assignment. The rationale is that, at this moment, the playback deadline is still far away (compared to Stage 2), and thus the network/computation-load uncertainty diminishes the benefits brought by a block-level, fine-grained search. Meanwhile, this reduces the solution space from $O(8^{wb})$ to $O(8^w)$. Specifically, we (1) start with a quasi-optimal solution obtained from an even coarser-grained search at the granularity of every two consecutive chunks, and (2) perform pruning by bounding [19]. After the above two optimizations, for a practical w (*e.g., w*=10), the search time (for maximizing the QoE in Eq. 8) becomes negligible compared to the downloading and upsampling time. To estimate I_i^{stall} in Eq. 8, at runtime, YuZu continuously estimates (1) the network bandwidth using the method in [29] and (2) the local processing time of a frame using EWMA-based estimation.

The second stage takes place *before upsampling each frame*. At this stage, the playback deadline gets closer and thus a *block-level, fine-grained search* would be beneficial. To reduce the search complexity, YuZu employs Simulated Annealing (SA) [40] – a probabilistic, greedy approach that approximates the global optimum. Our algorithm begins with setting all the blocks' SR ratios to the lowest (no SR). For each block, the algorithm tries to increase its SR ratio by one level. If the resulting QoE of the finite horizon increases, this change is always accepted; otherwise, we may still accept this change with a probability of $exp(-\frac{\Delta}{t})$, where Δ is the decrease of the QoE and t is the current number of iterations, to avoid a potential local maximum. To speed up the SA algorithm, we reduce the finite horizon to two frames: the previous frame and the current (to be upsampled) frame – we empirically find that conducting frequent adaptations with a short horizon at a per-frame basis outperforms infrequent adaptations with a long horizon at a per-chunk basis in terms of the QoE.

5.4 Coloring SR Results

As described in §2, none of the 3D SR models we investigated performs colorization. There are two high-level approaches for colorization. One is augmenting the SR models by adding the color component. This may yield good colorization results, but at the cost of significantly increasing the SR workload. Given this concern, YuZu takes a much more lightweight approach: approximating each upsampled point's color using the color of the nearest point in the low-density point cloud (*i.e.,* the input to the SR model). In Appendix C, we present the details of our method and experimentally confirm that it can indeed produce good visual quality (with a PSNR >38).

6 Implementation

We integrate all the components in §5 into YuZu, a holistic system as shown in Figure 1. Our implementation consists of 10,848 lines of code (LoC), with 8,326 LoC for the client.

For offline SR model training, we modify the source code of PU-GAN [10] and MPU [8] using TensorFlow 1.14 [13] and custom TensorFlow operators from SPH3D-GCN [12]. Our pre-trained models are saved in the ProtoBuf format [9] that is language- and platform-neutral, facilitating future reuse. For online streaming, we implement the client player on Linux in C++. We use the Draco Library [4] for encoding and decoding the point cloud data. We employ Bazel [3] to compile the TensorFlow 1.14 C/C++ library and use the compiled library to load and execute the SR models. The client *pipelines* content fetching (network-bound), point cloud decoding & patch generation (CPU-bound), 3D SR (GPU-bound), and colorization (CPU-bound) of different frames for better performance. The server is also built in C++, with a custom DASH-like protocol over TCP for client-server communication.

7 Evaluation

7.1 Experimental Setup

Volumetric Videos. We use four point-cloud-based volumetric videos throughout our evaluations. (1) Our own video. We capture a volumetric video by ourselves using 3 synchronized depth cameras. It has 3,622 frames (2 min) each consisting of ~100K points. We refer to this video as *Lab*. We have used it to motivate YuZu in §2. (2) The Long Dress (*Dress*) and *Loot* videos (§4.2). They have 300 frames (10 sec) each consisting of ~100K points. Since they are short, we loop them (with cold caches) 10 times in our evaluations. (3) The *Haggle* video (§4.2). It has 7,800 frames (4'20") each consisting of ~100K points. For all four videos, the eight possible resolution/SR-ratio assignments are listed in Table 3. For each video, we train their SR models separately. All the videos are at 30 FPS, encoded by Draco [4]. Unless otherwise mentioned, the results reported in the remainder of this section are generated using all four videos. The average encoded bitrate of *Lab*, *Dress*, *Loot*, and *Haggle* (4×1) are 96, 108, 118, and 118 Mbps, respectively.

M_1	The vanilla 3D SR model (PU-GAN and MPU)
M_2	M_1 and optimizing patch generation
M_3	M_2 and layer profiling & pruning
M_4	M_3 and applying the spherical kernal function (SKF)
M_5	M_4 and merging SR input with SR output
M_6	M_5 and caching/reusing SR results

Table 7: SR acceleration methods (cumulative).

3D SR Models. We apply our developed model acceleration techniques to two recently proposed 3D SR models: PU-GAN [43] and MPU [61]. The two models usually yield qualitatively similar results, so we show the results of PU-GAN by default. For certain SR-specific experiments (*e.g.,* SR acceleration), we show both models' results. The models are trained on a per-video basis. For each video, the total size of all its models (×2, ×3, and ×4) is around 1.25 MB.

Metrics and Roadmap. We thoroughly evaluate YuZu in terms of performance, QoE, and resource utilization. §7.2 evaluates the QoE improvement brought by our 3D SR optimizations using both subjective (*i.e.,* real-user ratings) and objective (*e.g.,* PSNR [30]) metrics. §7.3 focuses on the performance gain of our 3D SR optimizations, from the perspectives of resource usage, inference time, and upsampling accuracy. §7.4 and §7.5 evaluate the end-to-end performance (*e.g.,* QoE and data usage) of YuZu. §7.6 provides additional micro benchmarks.

Network Conditions. We consider the following network conditions that are readily available in today's wired and wireless networks. (1) *Wired network with stable bandwidth* (*e.g.,* 50, 75, and 100 Mbps) and ~10ms RTT. (2) *Fluctuating bandwidth* captured from real LTE networks. We collect 12 bandwidth traces from a major LTE carrier in multiple U.S. states at diverse locations (campus, malls, streets, *etc.*). Across the traces, their average bandwidth varies from 33.7 to 176.5 Mbps, and the standard deviation ranges from 13.5 to 26.8 Mbps. We use `tc` [6] to replay these traces (with a 50ms base RTT typically observed in LTE [38]). (3) We also conduct *live LTE experiments* at 9 diverse locations in a U.S. city where the average bandwidth varies from 41.1 to 52.4 Mbps and the standard deviation is between 16.6 and 20.7 Mbps.

Devices. We use a commodity machine with an Intel Core i7-9800X CPU @ 3.80GHz and 32GB memory as the YuZu server. We use three client hosts: (1) a desktop with an Intel Core i9-10900X CPU @ 3.70GHz, an NVIDIA GeForce RTX 2080Ti GPU, and 32GB memory (the default client used in our evaluations); (2) a desktop with the same CPU, an NVIDIA GeForce GTX 1660Ti GPU, and 32GB memory; (3) an NVIDIA Jetson TX2 embedded system board with a Pascal-architecture GPU of 256 CUDA Cores, 8GB memory, and a quad-core CPU. They represent a typical high-end PC, a medium-class PC, and a mobile device, respectively.

User Motion Traces. We collect 32 users' 6DoF motion traces when watching the four videos, and replay them in some experiments. The details about how we collect the motion traces can be found in Appendix B.
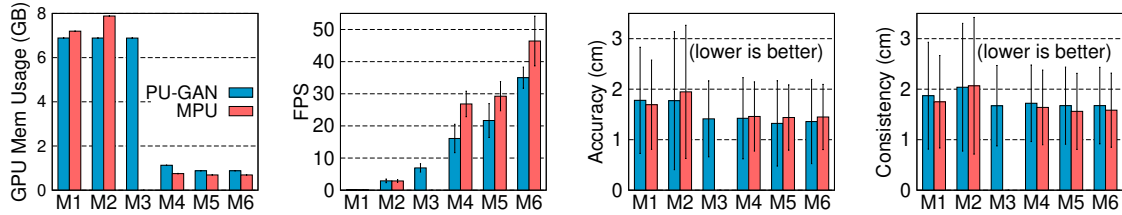
Figure 6: Memory usage, upsampling FPS, upsampling accuracy, and visual consistency of **M1** to **M6** (2080Ti desktop).
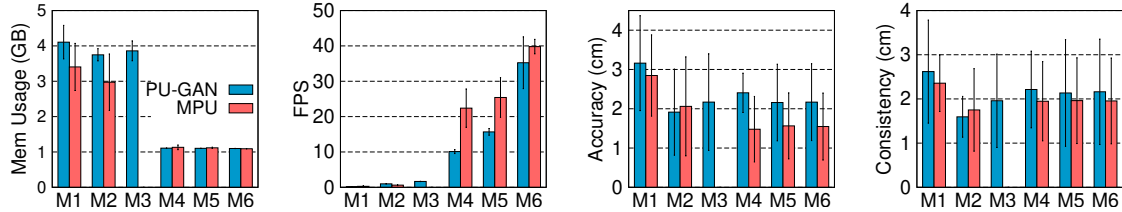
Figure 7: Memory usage, upsampling FPS, upsampling accuracy, and visual consistency of **M1** to **M6** (Jetson TX2 board).
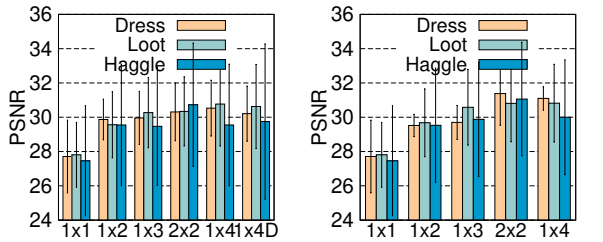
Figure 8: PSNR of YuZu (left) and vanilla PU-GAN (right).

7.2 SR Quality

Subjective Ratings. Recall that in our user studies, we ask our participants to rate the SR results generated by our optimized SR scheme (§5.1). Figure 15 shows that SR brings a significant boost to the user-perceived QoE. For example, at 1m, compared to 1×1, the user-rated QoE increases by 37%, 75%, and 150% for 1×2, 1×3, and 1×4, respectively; 2×2 improves the QoE by 178% compared to 2×1 (§4.2).

Objective Metric. We also examine how SR improves PSNR [30], an objective metric of image quality. The methodology is as follows. We replay the 32 users' 6DoF motion traces of watching the videos under different SR settings, and save the rendered viewports as images $\{I_{SR}\}$. We then repeat the above process using the original videos (4×1), and capture the viewport images $\{I_{4\times1}\}$. We compute the PSNR values by comparing each image in $\{I_{SR}\}$ with its corresponding image in $\{I_{4\times1}\}$. Figure 8 (left) shows the PSNR values for 1×1, 1×2, 1×3, 2×2, 1×4, and 1×4 with reusing SR results (denoted as "1×4D") across all the captured viewports. We notice a significant increase of PSNR from 1×1 to 1×2. The PSNR also increases marginally from 1×2 to 1×4. Meanwhile, the PSNR change between 1×4 and 1×4D is negligible, indicating that caching and reusing SR results brings little impact on the perceived video quality (but drastic performance gain as shown in §7.3). The results of *Lab* are similar. Note that a PSNR value over 30 typically indicates good visual quality [22, 58]. Figure 8 (right) shows the PSNR values for the unmodified PU-GAN model. The qualitatively similar results

between the left and right plots of Figure 8 indicate that our SR acceleration modifications sacrifice little visual quality. Note the above results include the colorization step, which is described and separately evaluated in Appendix C.

Comparing Figure 15 and Figure 8, we notice disparities between users' QoE ratings and PSNR values. This indicates that image qualities of rendered 2D content do not directly reflect the perceived QoE of volumetric content. This is a key reason for developing the QoE model for volumetric videos.

7.3 SR Performance Breakdown

We now take a closer look at the effectiveness of each of our proposed methods for accelerating SR. As listed in Table 7, M_1 denotes the vanilla 3D SR model as the comparison baseline; M_2 to M_6 are our proposed SR acceleration methods in §5.1 and §5.2. They are presented in a *cumulative* fashion, *i.e.,* M_i includes every feature of M_{i-1} plus some new feature. The experiments are conducted using two 3D SR models (PU-GAN [43] and MPU [61]), 100Mbps wired network, 4× SR, with network/compute resource adaptation (§5.3) disabled.

Figures 6 and 7 show the results of PU-GAN and MPU on the PC (2080Ti) and Jetson TX2 board, respectively. On the Jetson board, due to its low compute power (and mobile devices' small screen size), we reduce the original video's resolution from 100K to 20K points per frame (*i.e.,* the SR is from 5K to 20K points per frame). We consider four metrics: (1) maximum GPU memory usage (on Jetson TX2 we measure the system memory shared by GPU and CPU), (2) average upsampling speed (in FPS), (3) inference accuracy measured in EMD between each upsampled frame and the ground truth (4×1), and (4) visual consistency measured in EMD between each consecutive pair of upsampled frames.

As shown, on 2080Ti, for PU-GAN (MPU), compared to M_1, M_6 reduces the GPU memory usage by 87% (90%), accelerates the upsampling by 307× (542×), improves the average upsampling accuracy by 24% (14%), and slightly improves the consistency. Also, each optimization (M_2 to M_6) indi-
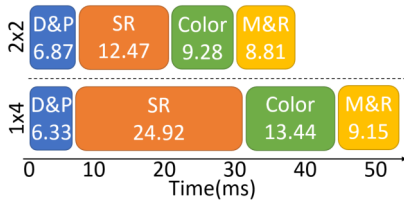
Figure 9: Frame processing time breakdown. D&P: decoding and patch generation; SR: upsampling; Color: colorization, M&R: merging and rendering.
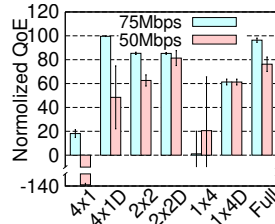
Figure 10: QoE over stable bandwidth ("D"=caching & reusing SR results).
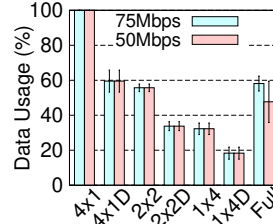
Figure 11: Data usage over stable bandwidth.
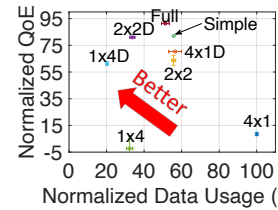
Figure 12: QoE vs. Data usage over fluctuating bandwidth (LTE traces).

Figure 13: QoE and data usage over live LTE networks.

Figure 14: YuZu over ViVo.

vidually improves the upsampling speed and possibly other metric(s). The Jetson setup shows a similar trend. The two models (PU-GAN and MPU) we studied exhibit similar performance gains as we progressively apply our optimizations, except that MPU is less sensitive to M_5. This is because of the network structure difference between PU-GAN and MPU. Note that we do not apply M3 to MPU because our layer-by-layer profiling (§5.1.1) reveals there is no layer that only makes a marginal contribution to the overall upsampling accuracy in the MPU model.

Latency Breakdown. Figure 9 shows the latency breakdown of processing an average frame using PU-GAN (*Lab* video, wired 100Mbps, 2080Ti desktop) under two settings: 2×2 and 1×4. As shown, SR remains the most time-consuming component. The breakdown for MPU is similar. The above results indicate the importance of SR acceleration.

7.4 Diverse Network Conditions

We evaluate the QoE of YuZu under different network conditions, using the four videos and the associated motion traces.

Stable Bandwidth. We first consider two stable bandwidth: 50Mbps and 75Mbps. Under each bandwidth profile, we run the full-fledged YuZu ("Full") and six statically configured YuZu instances: 4×1, 2×2, and 1×4 with and without SR result reusing. The QoE results are shown in Figure 10. We make several observations. First, when the bandwidth is low (50Mbps), 4×1 (without SR) gives the lowest (and even negative) QoE. This is because the limited bandwidth leads to high *network-incurred* stall when fetching high-resolution content; SR can effectively improve the QoE by using computation to compensate for the low bandwidth. Second, when the bandwidth increases to 75Mbps, 1×4 gives the lowest QoE due to the distortion and *computation-incurred* stall due to the high SR ratio. Instead, when the bandwidth is sufficient, the player should fetch the content with a higher quality (*e.g.,* 4×1D). Third, caching and reusing (C&R) the SR results improves the QoE when either the bandwidth is low (*e.g.,* 4×1 at 50Mbps), or the SR ratio is high (*e.g.,* 1×4). Under these two scenarios, C&R reduces the network and compute resource usage, respectively. The saved resources can be used to improve the content quality for other frames with more heterogeneity.

Figure 11 compares the (normalized) data usage, which is defined as the total downloaded bytes including the SR models and meta data. Compared to 4×1, applying C&R reduces the data usage by 40.5%. Also, increasing the SR ratio reduces the data usage, *e.g.,* 1×4D consumes only 18.3% of the data compared to 4×1. The full-fledged YuZu with adaptation gives the overall best QoE (Figure 10) and low data usage (Figure 11) by balancing the compute and network resource consumption. Compared to 4×1, full YuZu reduces the data usage by 52.3% (50Mbps) and 41.9% (75Mbps) while boosting the QoE by 214% (50Mbps) and 78.3% (75Mbps).
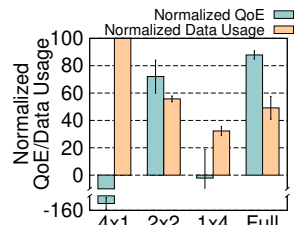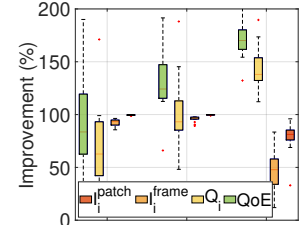
Fluctuating Bandwidth. We repeat the above experiment over fluctuating bandwidth emulated using our collected LTE traces (§7.1). The results are shown in Figure 12, which considers both the data usage (x-axis) and the QoE (y-axis). 4×1 yields the highest data usage; further applying C&R (4×1D) not only reduces the data usage by 40.5%, but also increases the QoE by 61.8% due to reduced stall. The full YuZu further improves the QoE by 21.0% and reduces the average data usage by 8.2%. This is achieved through strategically fetching lower-quality blocks and using higher SR ratios. In addition, the full YuZu improves the QoE by 10.4% to 93.7%, compared to 1×4 and 2×2 with and without C&R.

Live LTE. We conduct live LTE experiments at 9 locations in a major U.S. city. As shown in Figure 13, the results are largely aligned with those in Figure 12, except for the lower QoE of 4×1. This is because of the lower bandwidth of live LTE throughout the test locations compared to the LTE traces used in Figure 12. Compared to 4×1, the full YuZu improves the QoE by 210.3% and reduces the data usage by 50.8%.

7.5 YuZu vs. Existing Approaches

YuZu vs. Viewport-Adaptive Streaming. We compare YuZu with ViVo [27], a recently proposed viewport-adaptive approach. Leveraging 6DoF motion prediction, ViVo determines what content to fetch and which quality to fetch based on

predicted viewport and viewing distance. Similar viewport-adaptive approaches are used in the other systems [41, 50].

We develop a custom replication of ViVo on Linux in 7,101 LoC with the same set of configuration parameters. Figure 14 shows the improvement brought by YuZu compared to ViVo in terms of the overall QoE and its three components (Q_i, I_i^{patch}, and I_i^{frame}, see Eq. 8), using all four videos and the users' motion traces.[4] Note that both systems exhibit negligible stall so I_i^{stall} is not plotted. As shown, YuZu brings significant improvement on the average QoE (by 100.6% to 174.9%) and on each QoE component. YuZu outperforms ViVo due to three reasons. First, ViVo does not support SR, which YuZu leverages to boost the QoE. Second, ViVo's viewport adaptation approach becomes less effective when the whole scene appears inside the viewport (which oftentimes appears in our motion traces). SR does not suffer from this limitation. Third, to realize viewport adaptive streaming, ViVo has to perform 6DoF motion prediction, which is error-prone. In contrast, YuZu does not require motion prediction, and therefore exhibits more stable performance in particular when the motion is fast. Note that viewport-adaptation and SR are orthogonal approaches and can be *jointly* applied.

YuZu vs. Simple SR Adaptation. To demonstrate the efficacy of our network/compute resource adaptation design (§5.3), we compare it with a simple adaptation approach that differs in two aspects. First, unlike YuZu's *two-stage* adaptation, it only performs *single-stage* adaptation before downloading each chunk. Second, it employs a *deterministic* greedy algorithm that increases the SR ratio of each block within the finite horizon (in chronological order) until the QoE does not further improve. In contrast, YuZu employs a *probabilistic* greedy approach that is less vulnerable to a local maximum. We evaluate the simple adaptation algorithm using our LTE traces (§7.4) and plot its result as "Simple" in Figure 12. Compared to it, the full YuZu increases the average QoE by 11.4% and reduces the average data usage by 7.9%.

7.6 Micro Benchmarks and Resource Usage

We conduct experiments to show the following. (1) YuZu can work adaptively with different hardware (we compare the results on 2080Ti and 1660Ti; we also ported YuZu to an embedded system, see Figure 7). (2) The main memory (~5GB) and GPU memory (~2GB) usage of YuZu is acceptable. (3) The (one-time) offline training time is non-trivial but acceptable, and the sizes of SR models are negligible (<0.2% of the video size). The details can be found in Appendix D.

8 Related Work

Volumetric Video Streaming. There exist only a few studies on point-cloud-based volumetric video streaming [25–27, 31,

41, 50, 52, 59]. For example, DASH-PC [31] extends DASH to volumetric videos. PCC-DASH [59] is another DASH-based streaming scheme of compressed point clouds with bitrate adaptation support. ViVo [27] introduces visibility-aware optimizations for volumetric video streaming. GROOT [41] optimizes point cloud compression for volumetric videos. To the best of our knowledge, there is no existing work on applying 3D SR to volumetric video streaming.

Point Cloud SR. We can classify existing work on point cloud SR into two categories: optimization-based [16, 32] and learning-based [43, 61, 63, 70]. Most learning-based approaches follow the workflow established in PU-Net [70], which divides a point cloud into patches, learns multi-level point features of each patch, expands the features, and reconstructs the points from the features. All the above methods are designed for a *single* point cloud; they suffer from numerous limitations when applied to volumetric videos (§2).

Visual Quality Assessment of Point Clouds. The state-of-the-art visual quality assessment focuses on *static, non-SR* point clouds [23, 47, 60]. For example, using a data-driven approach, Meynet *et al.* [47] present a full-reference visual quality metric for colored point clouds. Different from the above studies, we model the QoE of SR-enhanced volumetric video streaming. We address new challenges on modeling the impact of various factors such as the viewing distance, upsampling ratio, and SR incurred distortion (§4).

SR for Regular 2D Videos. NAS [67, 68] is one of the first proposals that apply 2D SR to Internet video streaming. Other recent efforts on 2D SR include PARSEC [22] for 360° panoramic video streaming, LiveNAS [39] for live video streaming, and NEMO [66] for mobile video streaming. In contrast, YuZu addresses numerous unique challenges (§1) on applying 3D SR to volumetric video streaming.

9 Concluding Remarks

In this paper, we conduct an in-depth investigation on applying 3D SR to streaming volumetric content. Our proposed QoE model and the YuZu system take a first and important step toward making SR-enhanced volumetric video streaming principled, practical, and affordable. YuZu demonstrates how a series of novel optimizations, which fill a 500× performance gap, as well as judicious network/compute resource adaptation can help significantly improve the QoE for volumetric video streaming.

Acknowledgments

---

[4]ViVo does not have the notion of patch; instead its basic adaptation unit is a cubic cell. To ensure fair comparisons, we further divide ViVo's cells into virtual "patches" with the same size as YuZu and assign to them its parent cell's corresponding quality level when calculating I_i^{patch}.

References

[1] 8i Voxelized Full Bodies (8iVFB v2) - Dynamic Voxelized Point Cloud Dataset. http://plenodb.jpeg.org/pc/8ilabs.

[2] Amazon Mechanical Turk. https://www.mturk.com/.

[3] Bazel. https://bazel.build/.

[4] Draco 3D Data Compression. https://google.github.io/draco/.

[5] ITU-P.913: Methods for the subjective assessment of video quality, audio quality and audiovisual quality of Internet video and distribution quality television in any environment. https://www.itu.int/rec/T-REC-P.913.

[6] Linux TC Man Page. https://linux.die.net/man/8/tc.

[7] Magic Leap One. https://www.magicleap.com/en-us/magic-leap-1.

[8] MPU. https://github.com/yifita/3PU.

[9] Protocol Buffers. https://developers.google.com/protocol-buffers.

[10] PU-GAN. https://github.com/liruihui/PU-GAN.

[11] Qualtrics experience management platform. https://www.qualtrics.com/.

[12] SPH3D-GCN. https://github.com/hlei-ziyan/SPH3D-GCN.

[13] TensorFlow 1.14. https://github.com/tensorflow/tensorflow/tree/r1.14.

[14] The Octree Data Structure. https://en.wikipedia.org/wiki/Octree.

[15] Video Multimethod Assessment Fusion. https://en.wikipedia.org/wiki/Video_Multimethod_Assessment_Fusion, 2016.

[16] Marc Alexa, Johannes Behr, Daniel Cohen-Or, Shachar Fleishman, David Levin, and Claudio T. Silva. Computing and Rendering Point Set Surfaces. *IEEE Trans. on Visualization and Computer Graphics*, 9(1):3–15, 2003.

[17] Armen S Asratian, Tristan MJ Denley, and Roland Häggkvist. *Bipartite graphs and their applications*, volume 131. Cambridge university press, 1998.

[18] Athula Balachandran, Vyas Sekar, Aditya Akella, Srinivasan Seshan, Ion Stoica, and Hui Zhang. Developing a Predictive Model of Quality of Experience for Internet Video. In *Proceedings of ACM SIGCOMM*, 2013.

[19] Egon Balas and Paolo Toth. Branch and bound methods for the traveling salesman problem. 1983.

[20] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems*, 30(1-7):107–117, 1998.

[21] Hong Chang, Dit-Yan Yeung, and Yimin Xiong. Super-Resolution through Neighbor Embedding. In *Proceedings of CVPR*, 2004.

[22] Mallesham Dasari, Arani Bhattacharya, Santiago Vargas, Pranjul Sahu, Aruna Balasubramanian, and Samir Das. Streaming 360 degree Videos using Super-resolution. In *Proceedings of IEEE INFOCOM*, 2020.

[23] Rafael Diniz, Pedro Garcia Freitas, and Mylène C.Q. Farias. Towards a Point Cloud Quality Assessment Model Using Local Binary Patterns. In *Proceedings of the 12th International Conference on Quality of Multimedia Experience (QoMEX)*, 2020.

[24] Steven Fortune. Voronoi diagrams and delaunay triangulations. In *Comp. in Euclidean Geometry*, pages 225–265. World Sci., 1995.

[25] Serhan Gül, Dimitri Podborski, Thomas Buchholz, Thomas Schierl, and Cornelius Hellge. Low-latency cloud-based volumetric video streaming using head motion prediction. In *Proceedings of the 30th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video*, pages 27–33, 2020.

[26] Serhan Gül, Dimitri Podborski, Jangwoo Son, Gurdeep Singh Bhullar, Thomas Buchholz, Thomas Schierl, and Cornelius Hellge. Cloud Rendering-based Volumetric Video Streaming System for Mixed Reality Services. In *Proceedings of ACM MMSys*, 2020.

[27] Bo Han, Yu Liu, and Feng Qian. ViVo: Visibility-Aware Mobile Volumetric Video Streaming. In *Proceedings of ACM MobiCom*, 2020.

[28] Jian He, Mubashir Adnan Qureshi, Lili Qiu, Jin Li, Feng Li, and Lei Han. Rubiks: Practical 360° Streaming for Smartphones. In *Proceedings of ACM MobiSys*, 2018.

[29] Qi He, Constantine Dovrolis, and Mostafa Ammar. On the predictability of large transfer tcp throughput. *ACM SIGCOMM Computer Communication Review*, 35(4):145–156, 2005.

[30] Alain Hore and Djemel Ziou. Image quality metrics: PSNR vs. SSIM. In *Proceedings of the 20th International Conference on Pattern Recognition*, pages 2366–2369. IEEE, 2010.

[31] Mohammad Hosseini and Christian Timmerer. Dynamic Adaptive Point Cloud Streaming. In *Proceedings of ACM Packet Video*, 2018.

[32] Hui Huang, Shihao Wu, Minglun Gong, Daniel Cohen-Or, Uri Ascher, and Hao Zhang. Edge-Aware Point Set Resampling. *ACM Transactions on Graphics*, 32(1):9:1–9:12, 2013.

[33] Te-Yuan Huang, Ramesh Johari, Nick McKeown, Matthew Trunnell, and Mark Watson. A Buffer-Based Approach to Rate Adaptation: Evidence from a Large Video Streaming Service. In *Proceedings of ACM SIG-COMM*, 2014.

[34] Yan Huang, Jingliang Peng, C.-C. Jay Kuo, and M. Gopi. A Generic Scheme for Progressive Point Cloud Coding. *IEEE Trans. on Vis. and Computer Graphics*, 14(2):440–453, 2008.

[35] Erik Hubo, Tom Mertens, Tom Haber, and Philippe Bekaert. The Quantized kd-Tree: Efficient Ray Tracing of Compressed Point Clouds. In *Proceedings of IEEE Symposium on Interactive Ray Tracing*, 2006.

[36] Hanbyul Joo, Tomas Simon, Xulong Li, Hao Liu, Lei Tan, Lin Gui, Sean Banerjee, Timothy Scott Godisart, Bart Nabbe, Iain Matthews, Takeo Kanade, Shohei Nobuhara, and Yaser Sheikh. Panoptic studio: A massively multiview system for social interaction capture. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2017.

[37] Julius Kammerl, Nico Blodow, Radu Bogdan Rusu, Suat Gedikli, Michael Beetz, and Eckehard Steinbach. Real-time Compression of Point Cloud Streams. In *Proceedings of International Conference on Robotics and Automation*, 2012.

[38] Ali Safari Khatouni, Marco Mellia, Marco Ajmone Marsan, Stefan Alfredsson, Jonas Karlsson, Anna Brunstrom, Ozgu Alay, Andra Lutu, Cise Midoglu, and Vincenzo Mancuso. Speedtest-like measurements in 3g/4g networks: The monroe experience. In *Proceedings of the 29th International Teletraffic Congress (ITC 29)*, pages 169–177, 2017.

[39] Jaehong Kim, Youngmok Jung, Hyunho Yeo, Juncheol Ye, and Dongsu Han. Neural-Enhanced Live Streaming: Improving Live Video Ingest via Online Learning. In *Proceedings of ACM SIGCOMM*, 2020.

[40] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, 1983.

[41] Kyungjin Lee, Juheon Yi, Youngki Lee, Sunghyun Choi, and Young Min Kim. GROOT: A Real-Time Streaming System of High-Fidelity Volumetric Videos. In *Proceedings of ACM MobiCom*, 2020.

[42] Huan Lei, Naveed Akhtar, and Ajmal Mian. Spherical Kernel for Efficient Graph Convolution on 3D Point Clouds. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2020.

[43] Ruihui Li, Xianzhi Li, Chi-Wing Fu, Daniel Cohen-Or, and Pheng-Ann Heng. PU-GAN: A Point Cloud Upsampling Adversarial Network. In *Proceedings of ICCV*, 2019.

[44] Jyh-Ming Lien, Gregorij Kurillo, and Ruzena Bajcsy. Multi-camera tele-immersion system with real-time model driven data compression. *The Visual Computer*, 26(3):3–15, 2010.

[45] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural Adaptive Video Streaming with Pensieve. In *Proceedings of ACM SIGCOMM*, 2017.

[46] Rufael Mekuria, Kees Blom, and Pablo Cesar. Design, Implementation and Evaluation of a Point Cloud Codec for Tele-Immersive Video. *IEEE Trans. on Circuits and Systems for Video Technology*, 27(4):828–842, 2017.

[47] Gabriel Meynet, Yana Nehmé, Julie Digne, and Guillaume Lavoué. PCQM: A Full-Reference Quality Metric for Colored 3D Point Clouds. In *Proceedings of the 12th International Conference on Quality of Multimedia Experience (QoMEX)*, 2020.

[48] Carsten Moenning and Neil A Dodgson. Fast marching farthest point sampling. Technical report, University of Cambridge, 2003.

[49] Sergio Orts-Escolano, Christoph Rhemann, Sean Fanello, Wayne Chang, Adarsh Kowdle, Yury Degtyarev, David Kim, Philip Davidson, Sameh Khamis, Mingsong Dou, Vladimir Tankovich, Charles Loop, Qin Cai, Philip Chou, Sarah Mennicken, Julien Valentin, Vivek Pradeep, Shenlong Wang, Sing Bing Kang, Pushmeet Kohli, Yuliya Lutchyn, Cem Keskin, and Shahram Izadi. Holoportation: Virtual 3D Teleportation in Real-time. In *Proceedings of ACM UIST*, 2016.

[50] Jounsup Park, Philip A Chou, and Jenq-Neng Hwang. Volumetric media streaming for augmented reality. In *2018 IEEE Global communications conference (GLOBECOM)*, pages 1–6. IEEE, 2018.

[51] Charles R. Qi, Li Yi, Hao Su, and Leonidas J. Guibas. PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space. In *Proceedings of Conference on Neural Information Processing Systems (NeurIPS)*, 2017.

[52] Feng Qian, Bo Han, Jarrell Pair, and Vijay Gopalakrishnan. Toward Practical Volumetric Video Streaming On Commodity Smartphones. In *Proceedings of ACM HotMobile*, 2019.

[53] Feng Qian, Bo Han, Qingyang Xiao, and Vijay Gopalakrishnan. Flare: Practical Viewport-Adaptive 360-Degree Video Streaming for Mobile Devices. In *Proceedings of ACM MobiCom*, 2018.

[54] Yossi Rubner, Carlo Tomasi, and Leonidas J Guibas. The earth mover's distance as a metric for image retrieval. *International journal of computer vision*, 40(2):99–121, 2000.

[55] Paul Scovanner, Saad Ali, and Mubarak Shah. A 3-dimensional SIFT descriptor and its application to action recognition. In *Proceedings of the 15th ACM International Conference on Multimedia*, pages 357–360, 2007.

[56] Yiru Shen, Chen Feng, Yaoqing Yang, and Dong Tian. Mining Point Cloud Local Structures by Kernel Correlation and Graph Pooling. In *Proceedings of CVPR*, 2018.

[57] Steven L Tanimoto, Alon Itai, and Michael Rodeh. Some matching problems for bipartite graphs. *Journal of the ACM (JACM)*, 25(4):517–525, 1978.

[58] Nikolaos Thomos, Nikolaos V Boulgouris, and Michael G Strintzis. Optimized transmission of jpeg2000 streams over wireless channels. *IEEE Transactions on image processing*, 15(1):54–67, 2005.

[59] Jeroen van der Hooft, Tim Wauters, Filip De Turck, Christian Timmerer, and Hermann Hellwagner. Towards 6DoF HTTP Adaptive Streaming Through Point Cloud Compression. In *Proceedings of ACM Multimedia*, 2019.

[60] Irene Viola, Shishir Subramanyam, and Pablo Cesar. A color-based objective quality metric for point cloud contents. In *Proceedings of the 12th International Conference on Quality of Multimedia Experience (QoMEX)*, 2020.

[61] Yifan Wang, Shihao Wu, Hui Huang, Daniel Cohen-Or, and Olga Sorkine-Hornung. Patch-based Progressive 3D Point Set Upsampling. In *Proceedings of CVPR*, 2019.

[62] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. Image quality assessment: From error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004.

[63] Huikai Wu, Junge Zhang, and Kaiqi Huang. Point cloud super resolution with adversarial residual graph networks. In *arXiv preprint arXiv:1908.02111*, 2019.

[64] Francis Y. Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and Keith Winstein. Learning in situ: a randomized experiment in video streaming. In *Proceedings of USENIX NSDI*, 2020.

[65] Jianchao Yang, John Wright, Thomas S. Huang, and Yi Ma. Image Super-Resolution Via Sparse Representation. *IEEE Transactions on Image Processing*, 19(11):2861–2873, 2010.

[66] Hyunho Yeo, Chan Ju Chong, Youngmok Jung, Juncheol Ye, and Dongsu Han. NEMO: Enabling Neural-enhanced Video Streaming on Commodity Mobile Devices. In *Proceedings of ACM MobiCom*, 2020.

[67] Hyunho Yeo, Sunghyun Do, and Dongsu Han. How will Deep Learning Change Internet Video Delivery? In *Proceedings of ACM HotNets*, 2017.

[68] Hyunho Yeo, Youngmok Jung, Jaehong Kim, Jinwoo Shin, and Dongsu Han. Neural Adaptive Content-aware Internet Video Delivery. In *Proceedings of USENIX OSDI*, 2018.

[69] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. A Control-Theoretic Approach for Dynamic Adaptive Video Streaming over HTTP. In *Proceedings of ACM SIGCOMM*, 2015.

[70] Lequan Yu, Xianzhi Li, Chi-Wing Fu, Daniel Cohen-Or, and Pheng-Ann Heng. PU-Net: Point Cloud Upsampling Network. In *Proceedings of CVPR*, 2018.

[71] Anlan Zhang, Chendong Wang, Bo Han, and Feng Qian. Efficient volumetric video streaming through super resolution. In *Proceedings of the 22nd International Workshop on Mobile Computing Systems and Applications*, pages 106–111, 2021.

Appendices

A   Evaluation of QoE Gain Brought by SR

We study the QoE model for $q_{i,j}$ (Eq. 2) while keeping I_i^{patch}, I_i^{frame}, and I_i^{stall} as zero. This allows us to measure the impact of SR without interference from other factors.

We use the four videos introduced in §4.2 for the experiment. We apply our optimized PU-GAN algorithm (details

in §5.1) to perform upsampling, and create $\binom{8}{2}$ = 28 video clips where each clip contains 2 out of 8 versions in Table 3 side by side (in a random order). This approach is known as the double stimulus comparison scale (DSCS) method [5] as recommended by ITU (International Telecommunication Union). We repeat the above process for four viewing distances: 1m, 2m, 3m, and 4m, which are determined from a separate IRB-approved user study whose details are described in Appendix B. To maintain a fixed viewing distance d, we display the viewport at d meters in front of and facing the viewer. We generate 112 video clips at 4K resolution for each video segment.

Next, we design a survey using Qualtrics [11] and publish it on Amazon Mechanical Turk (AMT) [2]. In the survey, we invite each paid AMT subject to view the 112 clips of a random video segment (out of the 4 videos) in a random order. After watching each clip, the subject is asked to rate which side provides a better QoE through 7 choices ("left looks {much better, better, slightly better, similar to, slightly worse, worse, much worse} than right"). We have collected 512 subjects' responses with a total number of 57,344 ratings. We show the demographics of the participants in Table 2.

Figure 15 shows the average ratings of the 8 versions across all the users. The four subplots correspond to the four viewing distances. We make four observations. First, when the viewing distance is short, SR can effectively boost the QoE. For example, at 1m, compared to 1×1, the (user-rated) QoE increases by 37%, 75%, 150% for 1×2, 1×3, and 1×4, respectively; 2×2 improves the QoE by 178% compared to 2×1. Second, under the same point density, the upsampled version's QoE is usually lower than the original content's QoE, in particular when the SR ratio is large. This is caused by SR's distortion. However, the gap tends to reduce as the SR ratio decreases. Third, SR's gain diminishes as the distance increases, because the rendered object becomes smaller in the view. Note that the scores for different distances are not directly comparable. Fourth, the four video segments exhibit similar trends (figure not shown).

Converting User Ratings to Numerical Scores. For a given tuple of (user, viewing distance, video segment), we construct a weighted directed graph for the user based on his/her ratings, where the nodes are the 8 schemes. Assume a video clip contains schemes A (on the left) and B (on the right). If the user thinks that the left (right) is much better, better, or slightly better than the right (left), we add an edge from B to A (A to B) with a weight of 3, 2, and 1, respectively. If the user thinks that the left is similar to the right, we add two edges between A and B, one from A to B and the other from B to A, with both edges' weights set to 0. We then normalize the weights of all the edges to $[0,1]$ and apply the PageRank algorithm [20] to each graph to compute the weight of every node. We then use the weights (multiplied by 10 for easy interpretation) as the numerical scores of the 8 schemes for the corresponding (user, viewing distance, video segment)

tuple. Finally, for each of the 8 schemes under a given viewing distance, we average the numerical scores across all the tuples (of that viewing distance) to obtain the results shown in Figure 15. Note that for each viewing distance, the weights of all the schemes (in each of the graphs) add up to 1. As a result, the numerical scores of the same scheme for different viewing distances are not directly comparable.

B User Study for Collecting 6DoF Motion Traces

We conducted a separate IRB-approved user study for collecting 6DoF motion traces of volumetric videos. Specifically, it captured the viewport trajectories of 32 users who watched the four video segments (*Lab*, *Dress*, *Loot*, *Haggle*) introduced in §2 and §4.2 through either a mixed reality headset (Magic Leap One [7]) or an Android smartphone. We developed custom volumetric video players for both device types. The 6DoF motion data (yaw, pitch, roll, X, Y, Z) was captured at the granularity of 30 Hz. The participants are diverse in terms of their education level (from freshman to Ph.D.), gender (16 females), and age (from 22 to 57). We determine the viewing distances used in §4.2 by analyzing the above traces. As shown in Figure 16, about 70% of the viewing distances are less than 4m. Therefore, we set the maximum viewing distance to be 4m for our user studies, and select the other three distances by evenly dividing this maximum distance into four ranges (*i.e.,* at 1, 2, and 3m).

C Colorization Algorithm of YuZu and its Evaluation

Recall from §5.4 that YuZu takes a lightweight approach to color the SR results: it approximates each upsampled point's color using the color of the nearest point in the low-density point cloud (*i.e.,* the input to the SR model).

YuZu employs two mechanisms to speed up the nearest point search. First, the search is performed on an octree [14], which recursively divides a point cloud (as the root node) into eight octants, each associated with a child node. The levels of detail of the point cloud are controlled by the height of the tree. Performing nearest point search on an octree has a low complexity of $O(logN)$ where N is the number of nodes in the tree.

Second, YuZu caches and reuses the results of previously searched points. The cache is indexed by a point's discretized coordinates, and the cached value is the color looked up from the octree. When coloring an upsampled point, YuZu first performs cache lookup in $O(1)$; upon a hit, the cached color will be directly used as the color of the point; otherwise, YuZu performs a full octree search and adds the search result to the cache. The discretization granularity incurs a tradeoff between colorization performance and quality. We empirically observe that a discretization granularity of $1cm^3$ can yield good visual
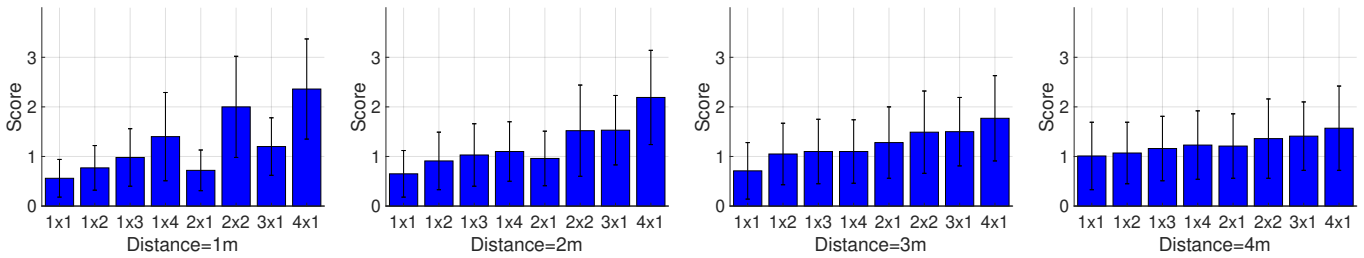
Figure 15: The average ratings of the 8 versions across all the users watching all the four video segments (*Long Dress*, *Loot*, *Band*, and *Haggle*).
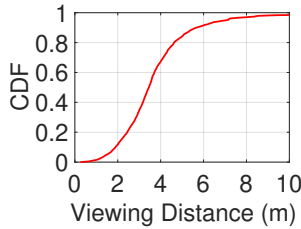
Figure 16: Distribution of viewing distance in our motion traces.

Figure 17: Impact of hardware and computation-aware adaptation.

quality under typical viewing distances ($\geq$ 1m).

We also notice opportunities for further improving the colorization quality. For example, the nearest point approach can be generalized into interpolating the nearest k points' colors; it can also be used in conjunction with DNN-based colorization, which may be more suitable for patches with complex, heterogeneous colors. Nevertheless, these enhancements inevitably increase the runtime overhead. We will explore them in future work.

Evaluation of Quality of Colorization. To evaluate the quality of the colorization step alone, we employ the approach in §7.2 where we use PSNR to objectively assess the image quality of rendered viewports. Specifically, we calculate the PSNR values by comparing $\{I_{4 \times 1}^{NP-Color}\}$ (defined below) with $\{I_{4 \times 1}\}$ (defined in §7.2), using the *Dress* and *Loot* videos and the real users' motion traces (Appendix B). The viewport images of $\{I_{4 \times 1}^{NP-Color}\}$ are obtained as follows: (1) remove the color from the original (4×1) video; (2) apply the above nearest-point (NP) colorization method to the video generated in Step (1), using the 1×1 video as the low-resolution point cloud stream from which the colors are picked; (3) replay the same motion traces to render the viewport images for the video colored in Step (2). The PSNR values of $\{I_{4 \times 1}^{NP-Color}\}$ are 38.09±2.44 and 44.15±2.59 for *Dress* and *Loot*, respectively, indicating the high fidelity of colors produced by our method. The above numbers are much higher than the PSNR values in Figure 8 (which also includes the colorization step) due to the following reason. PSNR and many other 2D image metrics such as SSIM [62] perform a pixel-wise comparison between two images. In the case of Figure 8, a tiny position shift of a 3D point may result in an also tiny position shift of its projected 2D pixel, leading to a pixel mismatch and thus a decreased PSNR score. This problem does not appear in the colorization step.
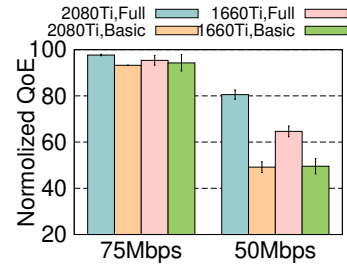
D Additional Micro Benchmarks

The following micro benchmark results are generated using the PU-GAN model. The results for the MPU model are qualitatively similar.

Impact of Computation-aware Adaptation. 3D SR demands considerable compute resources. Figure 17 demonstrates the impact of hardware and computation-aware adaptation, using the *Lab* video. Figure 17 considers two GPUs: a more powerful 2080Ti GPU and a less powerful 1060Ti GPU. It also considers two adaptation schemes: the full network/compute adaptation scheme described in §5.3 ("Full") and a computation-agnostic scheme that only adapts according to the network bandwidth ("Basic"). The Basic scheme works as follows. (1) It assumes that SR takes no time to complete; (2) it disables 2×2 and 1×4 (otherwise the QoE will degrade too much due to excessive stalls). Under the above setup, each bandwidth setting in Figure 17 has four schemes: {2080Ti, 1660Ti} × {Full, Basic}. As shown, when there is sufficient bandwidth, the QoE differences among the four schemes are small, because the player is more likely to fetch 3×1 and 4×1 blocks that do not require SR. However, when the bandwidth becomes low, the difference between 2080Ti and 1060Ti becomes noticeable, and the gap between Full and Basic is even larger. The Basic scheme yields much lower QoE scores because it ignores SR's computation overhead, leading to excessive stalls.

Memory Usage. We measure the client-side memory usage when streaming the *Lab* video over 50Mbps bandwidth (which leads to extensive invocations of SR). On the 2080Ti

(1660Ti) desktop, the peak main memory usage is 5.03GB (5.33 GB); the peak GPU memory usage is 1.97 GB (1.83 GB). YuZu's GPU memory usage on 2080Ti is higher than the numbers reported in Figure 6 because YuZu loads multiple SR models at runtime. When the available bandwidth is higher, the CPU/GPU memory will reduce because of fewer SR operations.

Offline Training Time and Model Size. YuZu incurs non-trivial model training time. For example, on the 2080Ti desktop, it takes about 88 minutes to train the 1×2, 1×3, and 1×4 models altogether for the *Lab* video consisting of 3,622 frames. However, note that (1) this is a one-time overhead; (2) we did not conduct any performance optimization for training; for a large-scale deployment, the training overhead could potentially be reduced by training one generic model and fine-tuning it for each specific video [68] (left as future work). The SR model size is negligible ($< 0.2\%$) compared to the video size.

NetVRM: Virtual Register Memory for Programmable Networks

Hang Zhu
Johns Hopkins University

Tao Wang
New York University

Yi Hong
Johns Hopkins University

Dan R. K. Ports
Microsoft Research

Anirudh Sivaraman
New York University

Xin Jin
Peking University

Abstract

Programmable networks are enabling a new class of applications that leverage the line-rate processing capability and on-chip register memory of the switch data plane. Yet the status quo is focused on developing approaches that share the register memory statically. We present NetVRM, a network management system that supports *dynamic* register memory sharing between multiple *concurrent* applications on a programmable network and is readily deployable on commodity programmable switches. NetVRM provides a *virtual register memory abstraction* that enables applications to share the register memory in the data plane, and abstracts away the underlying details. In principle, NetVRM supports *any* memory allocation algorithm given the *virtual register memory abstraction*. It also provides a default memory allocation algorithm that exploits the observation that applications have diminishing returns on additional memory. NetVRM provides an extension of P4, P4VRM, for developing applications with virtual register memory, and a compiler to generate data plane programs and control plane APIs. Testbed experiments show that NetVRM generalizes to a diverse variety of applications, and that its utility-based dynamic allocation policy outperforms static resource allocation. Specifically, it improves the mean satisfaction ratio (i.e., the fraction of a network application's lifetime that it meets its utility target) by 1.6–2.2× under a range of workloads.

1 Introduction

Programmable networks are a new paradigm that changes how we design, build and manage computer networks. Compared to traditional fixed-function switches, programmable switches allow developers to flexibly change how packets are processed in the switch data plane. The programming model of programmable switches are based on a multi-stage packet processing pipeline [8, 9].

Programmable switches provide different types of stateful objects that preserve states between packets, such as tables, counters, meters and registers. Among them, registers allow packets to read and write various states at line rate, which then affects how the following packets are processed. Such data-plane-accessible *register memory* is one of the defining features of programmable switches, and enables a new class of *reg-stateful* applications which utilize the on-chip register memory to realize various functionalities. These reg-stateful applications include not only the innovations in traditional network functions like congestion control [45], load balancing [25, 35] and network telemetry [1, 18], but also novel use cases beyond traditional networking, such as caching [23, 32], consensus [13, 14, 22] and machine learning [42, 43].

Given the rise of reg-stateful applications, an important open problem is how to support multiple concurrent reg-stateful applications running efficiently on a programmable network [51]. The utility of reg-stateful applications is usually decided by the amount of allocated register memory and the real-time network traffic [18, 23, 34, 47, 54, 58]. Thus, it is essential to dynamically allocate the limited register memory between multiple applictions to optimize the multiplexing benefits. Yet existing approaches of running multiple concurrent applications on programmable networks allocate register memory statically [19, 44, 49, 56, 57]. Changing the amount of register memory for one application would require recompiling and reloading the switch program, which would disrupt the operation of the switch.

In this paper, we propose NetVRM, a network management system that supports *dynamic* register memory sharing between multiple *concurrent* applications on a programmable network. NetVRM advances the status quo with three major features: The first one is a novel *virtual register memory abstraction*, which allows the register memory in the switch data plane to be dynamically allocated between multiple concurrent applications *at runtime*, without recompiling and reloading the data plane program. The second one is a dynamic memory allocation algorithm, which efficiently arbitrates the memory usage between concurrent applications based on the real-time utility measurements. The third one is a language extension and a compiler to generate data plane programs with the virtual register memory abstraction and efficient C++ control plane APIs for high-speed virtual register memory configuration.

The virtualization of register memory allows its dynamic allocation. Our approach is inspired by traditional virtual memory designs in operating systems, but programmable switches introduce two new challenges. First, register memory is distributed over multiple pipeline stages, and each register can be accessed only from one stage. Second, switch applications can access register memory from both the data

plane and control plane. NetVRM's memory system design is tailored to these characteristics. It places a page table at the front of the virtual register memory's processing pipeline, using it for memory translation in the data plane. The page table indexes the register memory regions allocated to each application in every stage. The switch control plane manages memory allocation. NetVRM also mediates application accesses to register memory from the control plane to ensure addresses are correctly translated.

NetVRM's dynamic memory allocation policy exploits the fundamental tradeoff between memory consumption and application utility. In particular, it leverages *diminishing returns*: the observation that, for most reg-stateful applications, the benefit of additional memory decreases with the amount of allocated memory [18, 23, 34, 47, 58]. For example, after a certain point, NetCache [23] cannot further improve the throughput significantly. More importantly, the memory-utility relationship changes both in the *temporal* and *spatial* dimensions based on application characteristics and traffic conditions. For example, the amount of register memory needed by NetCache depends on the request pattern, which can change over time and even vary across different switches. We design an online algorithm that does global memory allocation between applications in the network to maximize multiplexing benefits.

To make it easy to develop applications with NetVRM, we propose P4VRM, an extension to P4 [8]. P4VRM allows developers to virtualize register memory with a few simple modifications to existing P4 code: they mark register arrays to be virtualized and add online utility measurement primitives provided by P4VRM. The compiler takes multiple P4VRM programs as input and outputs a single P4 program with the virtual register memory abstraction and all the applications' functionalities, and generates the control plane APIs for high-speed virtual memory configuration.

In summary, we make the following contributions.

- We propose NetVRM, a network management system that exposes a virtual register memory abstraction to enable dynamic register memory sharing between multiple concurrent applications on a programmable network at runtime without recompiling and reloading.
- We design a dynamic memory allocation algorithm to efficiently allocate register memory between applications to maximize multiplexing benefits.
- We propose P4VRM, a data plane program extension, and provide a compiler to easily equip the data plane programs with virtual register memory and generate control plane APIs for efficient virtual memory configurations.
- We implement a NetVRM prototype. Testbed experiments on a variety of applications show that compared to static memory allocation, NetVRM improves the mean satisfaction ratio (i.e., the fraction of a network application's lifetime that it meets its utility target) by 1.6–2.2× under a range of workloads.

2 Motivation and Related Work

2.1 The Case of Dynamic Register Memory Allocation

Concurrent reg-stateful network applications. There are two broad types of objects provided by commodity programmable switches on the data plane—*stateless* objects, such as metadata, packet headers, and *stateful* objects, such as match-action tables, counters, meters, registers. Among them, registers, as one of the defining features of new-generation programmable switches, provide data-plane-accessible *register memory* for packets to read and write various states at line rate and enable much of the latest exciting research [14, 22, 25, 35, 42, 43, 45]. Register memory is implemented with standard SRAM blocks and can be read and written by both the control plane and data plane. Stateful Arithmetic and Logic Unit (ALU) performs register memory access and modification by executing a short program that involves register data, metadata and constant. The register memory is usually organized as register arrays. Each register array consists of several register slots with the same width and can be addressed by index (direct mapping) and hash (hash mapping). We refer to the network applications that use the register memory as *reg-stateful* applications.

Besides the rise and evolution of reg-stateful applications, modern cloud service providers usually serve multiple tenants concurrently [6, 30]. They allow tenants to run different network applications dynamically. For example, Azure and AWS provide a variety of network applications [5, 7] to their tenants, such as network address translation (NAT), load balancer, and network monitoring. We anticipate that the reg-stateful applications will be provided to tenants as programmable switches are being integrated in cloud networks, including both the datacenter networks and the wide area networks that connect the datacenter networks.

Necessity and potential benefits of network-wide dynamic allocation. The register memory on programmable switches is fundamentally limited by the hardware. For example, the maximal size of register memory on each stage is only a few Mb on the Intel Tofino switch [50]. Besides the limited register memory, there is a fundamental trade-off between memory consumption and application utility (e.g., its performance or accuracy) in many reg-stateful network applications [18, 23, 34, 47, 58]. Although some applications have a fixed memory requirement, most can operate with different amounts of available memory. Notably, our key observation is that applications generally exhibit *diminishing returns* [18, 23, 34, 47, 58]. The utility improvement decreases with more memory, and for many applications, additional memory has *no* utility after a point. We demonstrate the diminishing returns for four applications in Appendix A, including heavy hitter detection (HH) [54], newly opened TCP connection detection (NO) [55], superspreader detection (SS) [54] and NetCache [23]. The utility is measured using memory hit ratio (§5.1).

In all cases, the amount of memory affects the application utility, and such effects depend heavily on the workload. For example, NetCache [23] needs different amount of register memory with different skewed workload to deliver the same utility (Appendix A). Without dynamic allocation, this presents a formidable deployment challenge because the workload can vary in both temporal and spatial dimensions: different storage clusters see radically different workloads, and even a single cluster's request pattern changes over time (e.g., on a diurnal cycle) [4].

The diminishing returns and the temporally and spatially dynamic workload together also provide the opportunity to maximize resource multiplexing benefits by efficiently arbitrating the memory usage between concurrent applications.

2.2 Target and Scope of NetVRM

Target applications. The reg-stateful applications that can benefit from NetVRM must have the following properties.

- **They are elastic** (§5). An inelastic application (e.g., NetChain [22]) that has fixed virtual memory requirement can be supported by NetVRM, but cannot benefit from dynamic memory allocation.
- **The data plane programs have to meet the constraints in P4VRM** (§6), such as stateful ALUs since each operation of one register array must be associated with a specific stateful ALU.
- **The application utility should be obtained instantaneously** (§5.1). It can be computed on the switch (e.g., hit ratio as the default utility) or reported by applications.

We remark that there are a wide range of applications with the above properties, such as measurement applications [18, 39, 47], applications with approximate data structures [20, 34, 54], and caching applications [23, 33].

Register memory as the scope. There are a variety of resource types on a programmable switch, such as register memory, SRAM used for tables, TCAM and action units [51]. NetVRM focuses on dynamic allocation for register memory for three reasons. First, we observe that many reg-stateful applications are bottlenecked by register memory. Second, dynamic allocation of other resource types (e.g., match-action tables, TCAM) has been well-studied in the context of Software-Defined Networking (SDN) with traditional fixed-function switches [17, 21, 36, 46]. Third, current switch hardware cannot dynamically reallocate other resource types without rebooting the entire switch [51]. NetVRM is readily deployable on existing programmable switches.

Switch memory available that can be used as virtual register memory could be limited because a certain amount of memory has to be set aside for basic networking functionality, such as L3 routing, and inelastic applications (see §5). The evaluation in §8 shows that NetVRM outperforms the alternatives, regardless of *how much* physical memory is available for virtualization and dynamic allocation. Thus, NetVRM continues to be effective even as the memory for basic net-

working functionality and inelastic applications grows in size, leaving behind less memory for dynamic allocation.

2.3 Existing Solutions and Limitations

Recently, several existing works have explored how to support multiple applications on a programmable switch [19, 44, 48, 49, 56, 57]. At a high level, these solutions fail to meet the requirement of dynamic register memory allocation because of at least one of three limitations as follows.

- **Static binding of register memory.** Some of the existing work combine or merge multiple applications into one monolithic data plane program [19, 48, 56, 57] in compilation time. And the binding between register memory allocation and applications is static. Changing the allocation requires the data plane program to be recompiled and reloaded, during which the switch has to be stopped and restarted. This interrupts the operation of all applications on the switch, even the basic ones such as L3 routing.
- **Lack of a real switch environment.** Most of the existing solutions ignore the practical hardware constraints and are not applicable on a real ASIC-based switch (e.g., Intel Tofino [50]). For example, P4VBox [44] provides parallel execution of virtual switch instances on NetFPGA. MTPSA [49] realizes a multi-tenant portable switch architecture on NetFPGA and BMv2, a reference P4 software switch [3]. HyPer4 [19] and HyperV [56] realize the virtualization on software switches (e.g., BMv2, DPDK).
- **Not doing network-wide dynamic allocation.** Network resource allocation has been well studied for SDN with traditional fixed-function switches [16, 17, 21, 36, 37, 46]. For example, DREAM [36] does dynamic allocation for TCAM between measurement applications. However, none of the existing work has disclosed the potential benefit of a network-wide dynamic allocation for the *register memory* on programmable networks.

There are other related works that have explored how to manage and improve network applications on programmable networks. TEA [27] provides external DRAM for storing *table entries*, not *register memory*. Dejavu [52] utilizes the multiple pipelines and resubmission to fit a service chaining in one single switch. RedPlane [28] enables fault-tolerant stateful applications by designing a practical, provably correct replication protocol. NetVRM targets *register memory* and provides a new system for sharing it between multiple concurrent *reg-stateful* applications dynamically.

3 NetVRM Overview

NetVRM is a network management system that supports dynamic register memory sharing between multiple concurrent applications on a programmable network. Figure 1 shows an overview of NetVRM. NetVRM includes three critical components: virtual register memory, dynamic memory allocation and the P4VRM compiler. It abstracts away the complexities of allocating physical memory in each application, increases
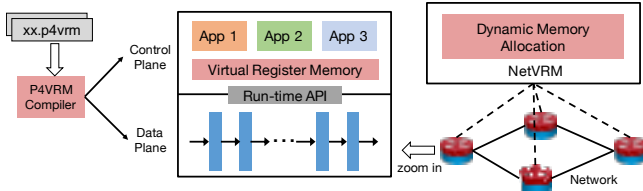
Figure 1: NetVRM overview.

memory utilization via statistical multiplexing, and provides P4VRM as an extension of P4 for developing applications with such virtual register memory.

Virtual register memory (§4). NetVRM exposes a virtual register memory abstraction to applications. The virtual register memory component in every switch hides the underlying details of the physical register memory that may span multiple stages and be shared with multiple applications. We design a custom data plane layout and an address translation mechanism to realize the virtual memory. The data plane layout composes the register arrays in multiple stages to one large register array, and allocates the large array to applications. Memory translation contains two page tables. One page table is in the data plane that translates the memory addresses computed from packet headers for memory access during packet processing, and the other is in the control plane for NetVRM to query and update the virtual memory of applications. The two tables are synchronized and managed by NetVRM.

Dynamic memory allocation (§5). In principle, NetVRM can support *any* memory allocation algorithm built on top of the virtual register memory. NetVRM also provides a default network-wide memory allocation algorithm for applications *without* knowing the utility functions. The algorithm exploits the diminishing returns between memory usage and application utility to maximize resource multiplexing benefits. We leverage the observation that many applications use the switch as a performance accelerator and deal with insufficient switch memory by having some kind of fallback path, either through the switch control plane or the servers [23, 29, 47]. As such, we cast the resource allocation problem as satisfying as many application's requirements as possible with respect to available memory size. This allows operators to specify application-specific utility metric and target for each application, avoiding the need to compare different utility functions across applications. NetVRM also provides a default, application-agnostic metric—the memory hit ratio—for applications that do not define their own.

Language extension and autogeneration (§6). NetVRM provides P4VRM, an extension to P4 [8] for developers to develop P4 programs with virtual register memory, and a P4VRM compiler to compose and compile individual P4VRM programs of different applications to one single P4 program with virtual register memory abstraction. The compiler also generates C++ APIs for efficient virtual register memory configuration in the control plane.
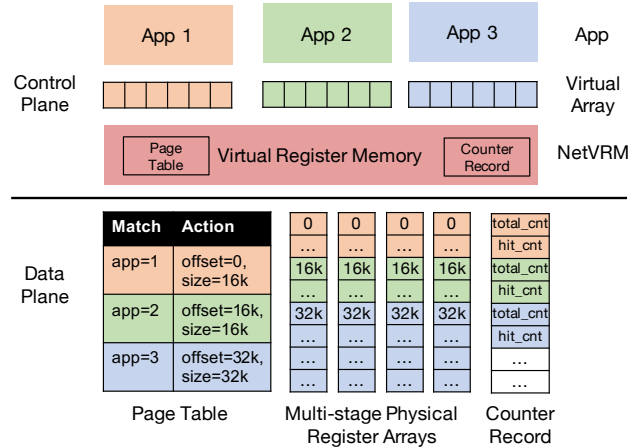
Figure 2: Virtual register memory design.

4 Virtual Register Memory

The register memory in the switch data plane is abstracted as register arrays for developers. The main problem of dynamically allocating memory is the coordination between multiple reg-stateful applications. Because register array definitions are hardwired in P4 programs, the code of an application has to be modified when other applications on the switch change, even if the application itself stays the same. NetVRM exposes a *virtual register memory space* to each application, which eliminates the coordination between applications. Each application is implemented with a virtual register array, without explicitly binding the register array to specific stages. As such, the application code does not need to be modified when the memory allocation changes. NetVRM is designed to manage the register memory and does not scarifice the support of recirculation.

Page table and counter record. A key challenge for memory virtualization on a switch, as opposed to a traditional CPU, is that the register memory can be accessed from both the data plane and the control plane (Figure 2). It is straightforward to implement the page table in the control plane. NetVRM simply does the translation in software. Specifically, it intercepts application memory accesses, uses the page table to perform the address translation, and then calls the memory access APIs of the switch driver to update the register memory configuration.

The page table in the data plane is more complicated, because it needs to be implemented using the programmable processing elements in the data plane. Figure 2 shows the design. The page table is implemented with a match-action table, and is placed at the stage before the physical register arrays to be virtualized. The match-action table matches on the application ID and identifies the location and size of the application's memory region (`offset` and `size`). These parameters are configured by the control plane at runtime as memory is allocated. We remark that *the page table does not introduce register memory overhead in common cases* (§7).

The counter record maintains two counters for each application, which only takes a small amount of memory. One is `total_cnt`, which tracks the total number of packets for an application. The other is `hit_cnt`, which tracks the number of packets that hit the switch register memory for each application. These counters are polled and reset periodically by the control plane to compute real-time memory hit ratios.

Memory layout. The memory layout partitions the physical register arrays *horizontally* across the stages. A virtual register array for an application is mapped to multiple blocks with the same start index (`offset` in the page table) and size (`size` in the page table) in each physical array. For example, in Figure 2 application 1 has a virtual array with 64K slots, which is mapped to $[0, 16K)$ in each physical array, and application 3 has a virtual array with 128K slots, which is mapped to $[32K, 64K)$ in each physical array.

This horizontal memory layout has three principal benefits. First, it decouples memory allocation from application code, and eliminates their static binding. The size of a virtual register array and its mapping to the physical arrays are represented by `offset` and `size` in action parameters, which can be dynamically changed at runtime, without recompiling and reloading the code in the data plane. Second, it enables fine-grained memory allocation. Because there are only a few stages (e.g., 10-20 stages) on commodity programmable switches [11, 50], our design can allocate the memory at row granularity (e.g., 8-slot granularity), which is fine-grained enough, compared with the total available slots on the switch (e.g., 512K). Third, it represents the memory layout using a small fixed-sized representation: only two variables (`offset` and `size`) per application. Although a more sophisticated memory layout might be able to achieve better space efficiency, more complex representations such as variable-length block lists would be challenging to implement efficiently in the data plane.

Address translation. Let the size of a virtual register array for an application be N. A virtual address $VA \in [0, N)$ is the index of the register slot in the virtual array. The physical address PA is computed by $PA = (VA/size, VA\%size + offset)$ after the page table, where $VA/size$ denotes the physical array index and $VA\%size + offset$ denotes the physical slot index in the corresponding stage. Division and modulo on arbitrary integers may not be supported in all switches. In such cases, we allocate virtual arrays with size to be a power of two, and implement these two operations with bit operations.

The above translation is sufficient for applications that directly access memory by VA. Besides these direct accesses, reg-stateful applications on programmable switches often use a lookup table or a hash function to access a register slot. Lookup tables use match-action tables to identify the address corresponding to a key (e.g., to find the memory location of an object in NetCache). We adapt the match-action table to hold a virtual address, then apply the VA to PA translation described above. Other applications use a

hash function to map a subset of header fields to a register slot (e.g., hashing the source IP in heavy hitter detection). While in principle the same translation approach can be used, hardware constraints on the Tofino platform mean that hash functions need to be associated with a particular address range, and adding a variable offset to the output requires an additional stage. NetVRM uses a hash function h_size, selected during the page table lookup stage, which has output in $[0, size)$. Hash lookups first compute $h\_size(pkt.hdr)$, then, in a subsequent stage, translate that to the physical slot location: $PA = (h(pkt.hdr)\%k, h\_size(pkt.hdr) + offset)$, where k is the number of physical arrays.

Some applications may need large virtual slots, each of which may be larger than a physical slot. In such cases, we combine multiple physical slots to implement a virtual slot.

5 Dynamic Memory Allocation

We classify reg-stateful applications on a programmable network into elastic and inelastic applications based on whether an application can work with a variable amount of register memory. An inelastic application requires a fixed amount of register memory; it cannot work with less (e.g., NetChain [22]). An elastic application does not have a fixed register memory requirement. Our key observation is that most elastic applications overcome insufficient register memory with a fallback mechanism to the network control plane or the servers [23, 47]. The amount of memory typically affects application-level performance metrics (e.g., the system throughout in NetCache [23]). Although it may be possible to transform inelastic applications to elastic ones [29], we leave that to application developers. NetVRM supports both types, while only elastic applications can benefit from NetVRM's dynamic memory allocation.

Each application is specified with four parameters: the *application type* (e.g., HH); the *subnet* in which the application will run (e.g., 10.0.0.0/8); the *utility metric*, which is either the default metric (i.e., memory hit ratio) or an application-specific one; and the *utility target* (e.g., 0.98 for memory hit ratio). For an inelastic application, the amount of required memory is specified instead of the utility metric and target. NetVRM allocates the memory to it if the requirement can be satisfied, and rejects the application otherwise.

Dynamic memory allocation is only performed for elastic applications. NetVRM periodically polls the counters from the data plane, obtains the utility of each application, and dynamically allocates the register memory between the applications based on their utilities. There is a long line of work related to network utility maximization [26, 38, 40]. NetVRM presents three particular challenges for network utility maximization, including how to define the application utility properly, how to approximate the utility functions, and how to allocate the register memory in the network, which will be demonstrated in detail as follows.
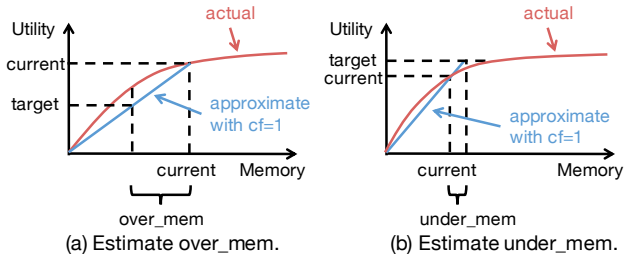
Figure 3: Utility function estimation.

Figure 4: Examples for network-wide allocation.

5.1 Definition of Application Utility

Finding a proper definition of application utility is challenging, because different applications have their own application-level objectives that cannot be directly compared with each other (e.g., accuracy for a heavy-hitter detector or throughput for NetCache). NetVRM allows applications to compute their own utility metrics and report them to the allocator. Because not all the application-level metrics can be reported online (e.g., accuracy for a heavy-hitter detector), NetVRM also provides a default, generic utility definition. It is based on the observation that for many elastic applications, a register memory miss in packet processing usually affects the application-level performance, e.g., extra latency to process a packet with the fallback mechanism. Therefore, one effective utility definition is the memory hit ratio, which is the ratio of packets directly processed by the register memory in the switch. Besides being application-agnostic, this utility can be computed by tracking counters for memory hits in the data plane by NetVRM itself (§4). Moreover, the memory hit ratio is also a widely-used metric to evaluate the workload reduction for the fallback mechanism in many elastic applications on programmable networks [18, 39, 47].

5.2 Problem Formulation

We denote the available virtual register memory size of c switches in the network as $M_1, M_2, ..., M_c$, respectively. There are l applications running in the network. Let $i.target$ be the utility target of application i, and $i.utility(i.m_1, ..., i.m_c, i.T)$ be the utility function of application i where $i.m_j$ is the memory usage of application i on switch j and $i.T$ is the real-time traffic of application i. The network resource allocation problem is formulated as follows.

$$max \sum_{i=1}^{l} \mathbf{1}(i.utility(i.m_1, ..., i.m_c, i.T) \geq i.target)$$

$$s.t. \sum_{i=1}^{l} i.m_j \leq M_j, \forall j = 1, ..., c$$

The objective is to maximize the number of applications of which the utility targets can be satisfied, and the constraint is to ensure the sum of allocated memory on each switch does not exceed its memory size. We remark that this is one objective that is provided by default and has been used in sev-

eral network management scenarios [36, 37]. NetVRM also supports other objectives and memory allocation algorithms.
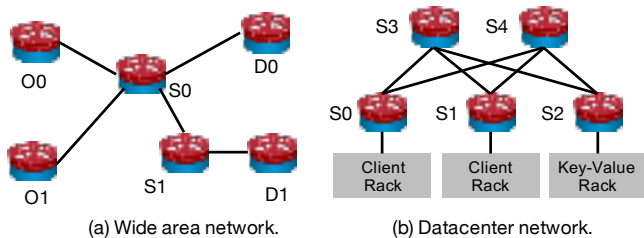
Main challenge: unknown and dynamic utility functions. The main challenge to solve the allocation problem is that the utility functions of the applications are unknown and change over time. It is true that some utility functions can be known as a priori, e.g., the worst-case accuracy and the memory requirement for sketch-based heavy hitter detection (SHH) using count-min sketch [12] can be calculated mathematically [54]. But utility functions for many applications such as HH, NO and SS (§2) are hard to know in advance. More importantly, the solution needs to adapt to real-time traffic and as the applications are started and stopped dynamically.

Solution: online utility curve estimation without application knowledge. In order to adapt memory allocation for the applications *without* knowing their utility function, NetVRM leverages the observation that the utility function follows diminishing returns, i.e., that it is concave, which holds for a wide range of reg-stateful network applications [18, 23, 34, 47, 58], and approximates the memory requirement for each application. Let $i.util$, $i.target$ and $i.mem$ be the current utility, the utility target and the current memory for application i, respectively. The utility function is approximated by a polynomial function that intersects the origin. For an application i above its utility target, we use

$$i.over\_mem \leftarrow i.mem - (\frac{i.target}{i.util})^{cf} * i.mem \quad (1)$$

to estimate the amount of memory that can be moved from i to other applications ($i.over\_mem$). Because of *diminishing returns*, the utility function is *concave* and a linear function (when $cf = 1$) may underestimate $i.over\_mem$ (Figure 3(a)). We use a compensation factor cf which is set to be larger than 1 to compensate this. For an application i below its utility target, we use

$$i.under\_mem \leftarrow (\frac{i.target}{i.util})^{cf} * i.mem - i.mem \quad (2)$$

to estimate the amount of memory to be added to i ($i.under\_mem$). We use a cf larger than 1 for faster convergence (Figure 3(b)).

5.3 Network-Wide Register Memory Allocation

Based on the approximation in §5.2, NetVRM uses an online algorithm to move memory from over-provisioned applications (those above their utility targets) to under-provisioned applications (those below their utility targets) to maximize

Algorithm 1 Network-wide memory allocation

```
1:  new_plan ← cur_plan.copy()
2:  for application i in applications do
3:      if i.util ≥ i.target then
4:          satisfied_list.append(i)
5:          i.over_mem ← i.mem − (i.target/i.util)^cf * i.mem
6:          distributed i.over_mem to i.paths proportionally
7:      else
8:          unsatisfied_list.append(i)
9:          i.under_mem ← (i.target/i.util)^cf * i.mem − i.mem
10:         distributed i.under_mem to i.paths inverse proportionally
11: sort satisfied_list by over_mem in decreasing order
12: sort unsatisfied_list by i.under_mem in increasing order
13: for application i in unsatisfied_list do
14:     for path p in i.paths do
15:         sort p.switches based on i's existence and s.over_mem
16:         for switch s in p.switches do
17:             allocate memory from satisfied_list to p.under_mem
18:     if all paths are satisfied then
19:         update new_plan
20:     else
21:         move memory back to satisfied_list
22: return new_plan
```

the objective. The allocation are performed periodically to handle real-time traffic dynamics and application changes.

Main challenge: multiple and overlapped paths of an application. Besides the unknown and dynamic utility functions, the network-wide allocation problem is further complicated by the following two challenges. First, an application may need to handle traffic between multiple origin-destination (OD) pairs, and the traffic between each OD pair may use multiple paths. For example, in a wide area network, the operator may want to detect heavy hitters for flows between multiple OD pairs, e.g., O0-D0 and O1-D1 in Figure 4(a). In a datacenter network, the operator may want to provide in-network caching for traffic from multiple client racks to a key-value store rack, e.g., S0-S2 and S1-S2 in Figure 4(b). Datacenter networks typically use multi-path routing, e.g., path S0-S3-S2 and path S0-S4-S2 for traffic between S0 and S2. Second, different paths of an application may overlap, and thus can share their allocated memory. For example, in Figure 4(b), NetCache can be placed in S2 to save memory instead of in both S3 and S4.

Solution: network-wide memory allocation. At a high level, NetVRM performs network-wide memory allocation in two steps. First, NetVRM uses the utility estimation mechanism in §5.2 to estimate the required memory for each application, and decomposes *over_mem* or *under_mem* of each application to multiple paths. Second, it moves the memory from over-provisioned applications to under-provisioned applications. The pseudocode is shown in Algorithm 1.

The first step is to compute and decompose *over_mem* or *under_mem* of each application to multiple paths (line 2-10). NetVRM measures the utility (i.e., the memory hit ratio by default) and the traffic on each path. With the memory hit ratio as the utility, the utility (memory) of application *i* is the weighted average of its utilities (memories) by the traffic

developers
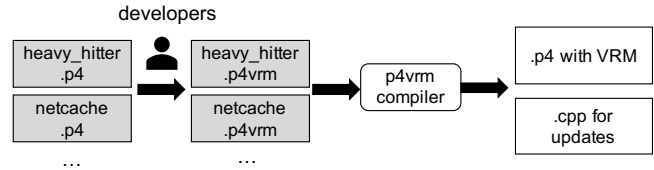
Figure 5: P4VRM compiler compiles P4VRM programs.

volume on its paths. We use the utility estimation mechanism in §5.2 to estimate *i.over_mem* and *i.under_mem*. Then *i.over_mem* is distributed to each path in proportional to their traffic (line 6) and *i.under_mem* is distributed to each path in inverse proportional to their traffic (line 10). We remark that NetVRM also allows disproportional memory allocation.

The second step is to move memory from over-provisioned applications to under-provisioned applications (line 11-21). We use a heuristic that reduces the memory for applications that are more over-provisioned first, and allocates the memory to the applications that are more likely to be satisfied first (line 11-12). For each unsatisfied application, it tries to satisfy the estimated memory requirement on each path (line 13-21). Because each path contains several switches, the algorithm needs to decide which switch to allocate memory from to satisfy the application (line 15-17). Two factors are considered in the decision, which are whether the application already has memory allocated on a switch (i.e., *i*'s existence) and how much extra memory a switch has (i.e., *s.over_mem*). These factors aim to avoid small amounts of memory scattering in many switches. If the application's requirement can be satisfied, the plan is updated (line 18-19). Otherwise, the memory is moved back to the satisfied applications (line 20-21).

To accommodate path overlapping, two extensions are required to the algorithm. First, in the utility estimation, the memory on overlapping switches is counted once for each overlapping path. Second, in memory allocation, the memory allocated to an application on overlapping switches is also counted once for each overlapping path.

Admission control, drop and priority. Admission control is critical when the total memory requirement exceeds the register memory size in the network. NetVRM admits one application into the network only if there is more available memory on each path than a predefined fraction of the total memory. NetVRM drops one application if it cannot meet the utility target in multiple consecutive allocation epochs. NetVRM targets elastic applications which can work even with no register memory. Thus, *if one application is rejected or dropped, it can turn to the fallback mechanism.* A malicious application with a tough utility target to satisfy would likely be dropped after a few allocation epochs. The operator can also assign custom priorities for the applications. For example, an application can be configured to not be dropped, or be assigned with a minimal amount of memory to avoid starvation when it is under-provisioned.

$\langle p4\_declaration \rangle ::= \langle vrm\_reg\_declaration \rangle \mid \langle vrm\_blb\_declaration \rangle \mid ...$

$\langle vrm\_reg\_declaration \rangle ::=$ 'vrmReg' $\langle virt\_stage \rangle \langle register\_declaration \rangle$

$\langle vrm\_blb\_declaration \rangle ::=$ 'vrmMergeable' $\langle blackbox\_declaration \rangle$
   $\mid$ 'vrmNonMergeable' $\langle blackbox\_declaration \rangle$

$\langle table\_declaration \rangle ::= ...$
   $\mid$ 'vrmMergeable' $\langle virt\_stage \rangle \langle table\_declaration \rangle$
   $\mid$ 'vrmNonMergeable' $\langle table\_declaration \rangle$

$\langle action\_function\_declaration \rangle ::= ...$
   $\mid$ 'vrmMergeable' $\langle action\_function\_declaration \rangle$
   $\mid$ 'vrmNonMergeable' $\langle action\_function\_declaration \rangle$

$\langle control\_statement \rangle ::= ...$
   $\mid$ 'HIT_COUNTER;'
   $\mid$ 'PKT_COUNTER;'

$\langle virt\_stage \rangle ::= \langle decimal\_value \rangle$

Figure 6: The P4VRM extensions to the P4-14. Gray nonterminal nodes refer to legacy rules in P4-14.

Memory reallocation process. At the end of each allocation epoch, NetVRM fetches the counters from the control plane, and computes the online utilities and the new memory allocation plan. Updating the memory allocation plan results in remapping from virtual addresses to physical addresses and moving existing entries because of the remapping. There are general solutions that can be applied to ensure the consistency of memory allocation updates [24, 53]. We apply two optimizations for particular cases in NetVRM. First, network measurement applications periodically reset the state such as counters maintained by the register memory. We align the memory allocation updates with the resetting operations, so that the memory allocation can be updated without moving existing entries and does not scarifice application correctness. Second, network applications that use lookup-table-based address translation can simply use a delta update when the memory size decreases, and allow more entries when the memory size increases. This ensures consistency because a lookup table is used for maintaining each address mapping.

6 Language Extension and Autogeneration

NetVRM provides P4VRM, an extension to the basic syntax and semantics of the P4 programming language [8] that supports virtual register memory abstraction and online utility measurement. Our implementation is based on P4-14, as more existing implementations are implemented in this version, but the same extensions could be applied to P4-16 as well. As shown in Figure 5, to port existing *.p4* programs, developers extend them to *.p4vrm* programs by marking which register arrays are to be virtualized and adding the online utility measurement primitives (HIT_COUNTER and PKT_COUNTER) correctly according to the applications. The P4VRM compiler takes multiple *.p4vrm* programs as input and outputs one merged P4 program (for the data plane) with virtual memory abstraction and online utility measurement, together with the C++ APIs (for the control plane) to configure the virtual register memory efficiently.

```
+ #include "params.p4"
- vrmReg 1 register stg1 {
+ register virtual_stg1 {
    width:32;
-   instance_count:8192;
+   instance_count:65536;
  }
- vrmNonMergeable blackbox stateful_alu salu_stg1 {
+ blackbox stateful_alu salu_stg1 {
-   reg: stg1;
+   reg: virtual_stg1;
    ...
  }
- vrmNonMergeable action act_stg1() {
+ action act_stg1() {
-   salu_stg1.execute_stateful_alu_from_hash(hash_1);
+   salu_stg1.execute_stateful_alu(params_md.slot_idx);
  }
- vrmNonMergeable table tbl_stg1 {
+ table tbl_stg1 {
+   actions {act_stg1;};
+   default_action:act_stg1();
+ }

  control ingress {
    if (valid(tcp) or valid(udp)) {
+     apply(set_app_id);
+     apply(set_offset_hf);
+     apply(add_offset);
+     if (params_md.app_type==0) {
        apply(tbl_stg1);
        ...
-       HIT_COUNTER;
+       apply(hit_counter);
        ...
-       PKT_COUNTER;
+       apply(pkt_counter);
+     }
    }
  }
```

Figure 7: An example of P4VRM code transformation by P4VRM compiler. '-' and '+' annotate the change before and after the transformation, respectively.

Grammar. As shown in Figure 6, P4VRM extends the P4-14 language specification [2] by introducing new keywords (vrmReg, vrmMergeable and vrmNonMergeable) to tag declarations related to a register array (register, blackbox, action, and table). It marks the register array as virtualized, and marks the related blackboxes, actions and tables that have the same logic as mergeable. It also specifies the stages at which the mergeable tables should be placed (virt_stage). The two primitive statements (i.e., HIT_COUNTER and PKT_COUNTER) are used for online utility measurement. HIT_COUNTER tracks the number of packets processed by the register memory, and PKT_COUNTER tracks the total number of packets of the application.

Generating merged P4 programs and C++ APIs. To merge parsers, P4VRM compiler abstracts the packet parser of each application as a Finite State Machine (FSM) and merges the identical states into a single FSM. Then, the P4VRM compiler transforms P4VRM-introduced declarations (i.e., vrmReg, vrmMergeable and vrmNonMergeable) to P4-14 declarations (i.e., register, blackbox, action and table), and adds the additional logic for address trans-
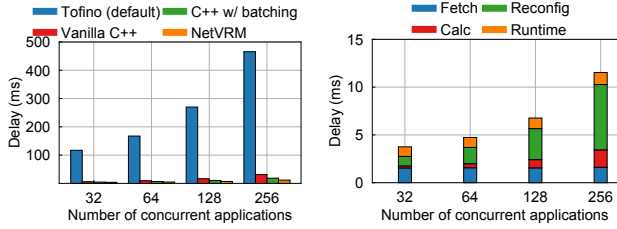
(a) Total control loop delay vs. differ- (b) Delay breakdown for NetVRM.
ent implementations.

Figure 8: Analysis of control loop delay.

lation, as shown in Figure 7. The compiler also loads a P4 library (`params.p4`) provided by P4VRM, containing additional metadata and logic (e.g., to perform the page table lookup) and adds the appropriate invocations at the beginning of the pipeline. Finally, the compiler generates control plane APIs for resetting counters, fetching counters, resetting virtual memory and configuring the virtual memory.

Requirement for merge. Merging multiple reg-stateful applications needs to comply with the same resource constraints as in existing work [19, 56, 57], most notably those related to register memory (e.g., total register memory size per stage, stateful ALUs per stage). If merging violates hardware constraints, the P4VRM compiler would fail and produce no output.

7 Implementation

We have implemented a NetVRM prototype on a 6.5 Tbps Intel Tofino switch [50], and used commodity servers to replay traces and generate traffic. The P4 library we provide for virtual register memory support is around 500 lines of P4-14 code. The virtual register memory spans eight physical stages. Other stages are used for necessary functionalities (e.g., routing and enabling concurrent applications). We emulate four switches with the four independent pipelines of the Tofino switch. The data plane program decides which emulated switch one packet enters by checking the ingress port. The implementation batches the data plane updates, and uses multithreading to update the four pipelines simultaneously. The NetVRM control plane implementation consists of around 2200 lines of C++ code. The P4VRM compiler is built on Flex/Bison [31] and parses the .p4vrm files to build an AST. It consists of around 2000 lines of C++ and 900 lines of grammar.

Overhead of NetVRM. The address translation needs to be done in two stages (§4), which is realized with two tables to adjust the `slot_idx` (Figure 7). The first table (`set_offset_hf`) can be placed in the same stage with other tables (e.g., `set_app_id`) that are necessary and inevitable for concurrent applications running. The register memory in the second stage where `add_offset` is placed cannot be virtualized, which can be used for basic networking functionality and inelastic applications. In some cases, the register memory in some stages cannot be used even without
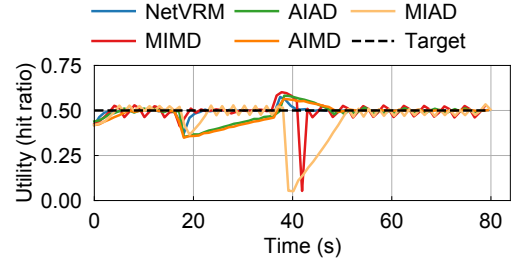
(a) Application utility over time.
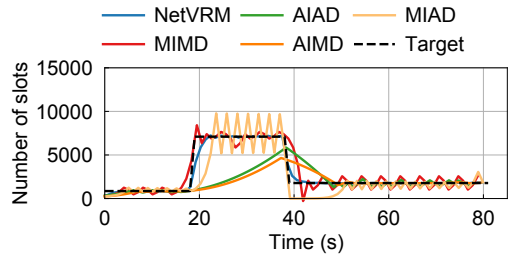
(b) Register memory consumption over time.

Figure 9: Comparison of different algorithms to update memory allocation.

NetVRM because of the indivisibility between the virtual slot size and the number of stages. For example, if there are three physical stages available for virtualization, an application with 2-stage virtual slots can use two stages at most. Then the page table placed in the first stage does not introduce extra register memory overhead. We remark that this is a common case for many applications [18, 23, 34, 47, 54]. The extra resource needed by each application in NetVRM is only one table entry in the page table and two counters for the online utility measurement, without extra stage overhead.

8 Evaluation

We evaluate our NetVRM prototype in two scales. We first use microbenchmarks to examine the control loop delay and the properties of the resource allocation algorithm (i.e., stability and convergence speed). With macrobenchmarks, we demonstrate the benefits of NetVRM in combination with a variety of network applications, workload parameters, comparisons with alternative approaches and network topologies.

8.1 Microbenchmark

Control loop delay. We emulate four switches by the four independent pipelines of the Tofino switch. First, we compare the total control loop delay, i.e., the time to complete a virtual memory reallocation (§5.3), with different implementations, including the default implementation on Tofino switches which uses Python Thrift APIs, a vanilla C++ implementation, a C++ implementation with batching, and NetVRM, which incorporates both batching and multithreading. As shown in Figure 8(a), the C++ implementations are an order of magnitude more efficient than the default implementation of Tofino control plane APIs. NetVRM's optimizations further reduce the delay by a factor of ~ 3.

(a) Heavy hitter detection (HH).   (b) Newly opened TCP (NO).   (c) Superspreader detection (SS).   (d) Mix of HH, NO, SS.
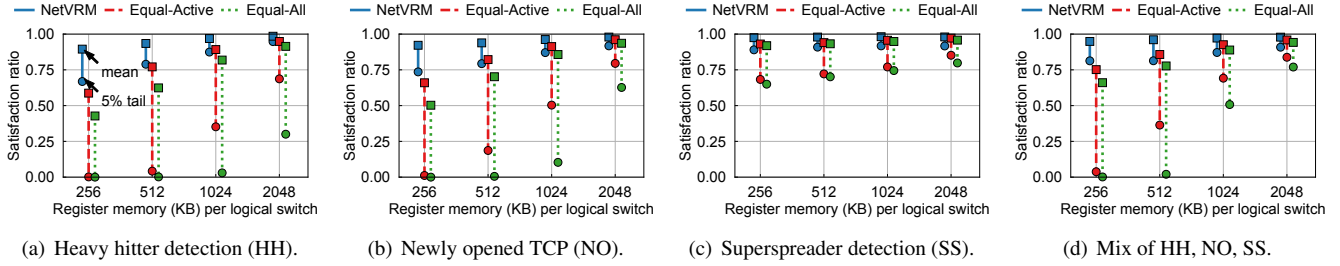
Figure 10: Satisfaction for flow-based applications in the WAN scenario.
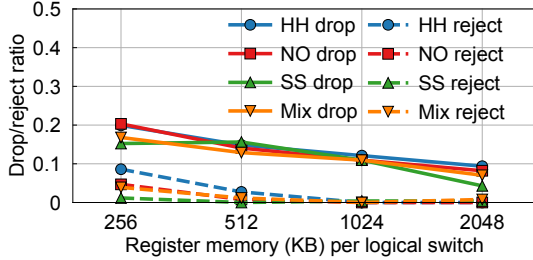
Figure 11: Drop/reject ratio of NetVRM for flow-based applications in the WAN scenario.

We further break down the control loop delay of NetVRM into four parts, i.e., *Fetch*, *Calc*, *Reconfig* and *Runtime*, and measure the latencies with different number of concurrent applications. *Fetch*, *Calc*, *Reconfig* and *Runtime* represent the time of fetching counters, calculating online utility and new memory allocation plan, configuring the page table, and the runtime overhead for resetting the state (e.g., the counters), respectively. As shown in Figure 8(b), the time of *Fetch* remains relatively constant since we use batching to fetch all the counters together where the data size does not influence the latency significantly. The time of *Calc* increases with more applications, due to the heavier overhead to compute the online utility and memory allocation plans. The time of *Reconfig* dominates the control loop delay because of the intensive updates to the data plane for four pipelines.

Due to the limit of our testbed, we only emulate four switches with one Tofino switch in our experiment. We remark that NetVRM can maintain the low control loop delay and scale in real wide area networks and datacenters with a larger number of switches for two reasons. First, *Fetch*, *Reconfig* and *Runtime*, which do not need coordination between multiple switches, can be done in different switches locally and simultaneously. Second, *Calc* needs to compute the online network-wide utility and memory allocation plans for each application which has to be done in a centralized location. Instead of doing it on the switch OS with limited computation capability in our experiment, the time of *Calc* can be reduced easily by running it in a more powerful server.

Stability and fast convergence of NetVRM. In this experiment, we compare NetVRM with other alternative approaches which are commonly used in network resource allocation, including AIAD, MIAD, MIMD and AIMD. Those approaches

estimate the memory requirements by increasing (decreasing) the step size additively (A) or multiplicatively (M) when the satisfaction status remains the same (changes) compared with the previous epoch. We run one NetCache [23] application on the switch and set its memory hit ratio target to be 0.5. The workload skewness is Zipf-0.99 at the beginning, then changes to Zipf-0.9 at 18 seconds, and finally changes to Zipf-0.95 at 38 seconds. Figure 9(a) and Figure 9(b) show the utility and memory usage over time, respectively. AIAD and AIMD fail to meet the utility target when the skewness becomes Zipf-0.9 because increasing the memory additively is too slow. MIAD converges slower after 38 seconds because decreasing the step size additively from a large step size is slow. MIMD has the closest performance to NetVRM, but the utility fluctuates around the utility target after convergence. NetVRM estimates the memory requirements based on the online utility (§5.2). Thus, it can react fast and more accurately to the traffic dynamics and maintain the utility above its target most of the time.

8.2 Macrobenchmark

NetVRM configuration and network topology. The default allocation epoch and measurement epoch are both one second. The default network topology is the Wide Area Network (WAN), where each application has traffic from 4 switches independently. NetVRM drops an application if it cannot meet the utility target in four consecutive epochs and rejects an application if the available memory on the switch is smaller than 1/128 of the total memory.

Network applications. NetVRM supports a wide range of network applications. We use five applications in the evaluation, i.e., heavy hitter detection (HH) [47], newly opened TCP connection detection (NO) [55], superspreader detection (SS) [47], sketch-based heavy hitter detection (SHH) [54] and NetCache [23]. HH, NO and SS are flow-based applications which store precise flow records on the data plane, and evict the existing entries to the control plane upon hash collisions, following the eviction policy in TurboFlow [47]. SHH is a sketch-based application that uses approximate data structures (i.e., count-min sketch [12]) to approximate flow records. NetCache maintains hot key-value pairs on the data plane to serve a request upon a cache hit. For each application type, there can be multiple instances of this application, e.g., belonging to different clients/tenants. Each client/tenant owns
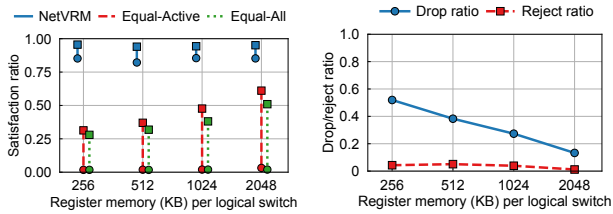
(a) Satisfaction.   (b) Drop/reject ratio.

Figure 12: Experimental results for sketch-based applications (SHH) in the WAN scenario.

a /8 subnet of source IP, and can dynamically start or stop application instances within its subnet.

Traffic traces. The traces for measurement applications on WAN are the 2019 passive CAIDA traces [10]. The datacenter traces are from Facebook's production clusters [41]. We replay the traces via MoonGen [15]. The NetCache traffic is generated by our DPDK client according to the Zipf distribution with different skewness parameters.

Alternative approaches. We compare NetVRM with two alternative approaches. (*i*) One is *Equal-All*, which *statically* assigns an equal amount of register memory to all applications, active or not. For example, if each application instance runs within a /8 subnet, then there are at most 256 concurrent application instances. Thus, *Equal-All* assigns 1/256 of total memory to each instance. (*ii*) The other is *Equal-Active*, which only assigns an equal amount of register memory to *active* instances. We emphasize that *Equal-Active* is enabled by the ability of NetVRM to dynamically allocate register memory at runtime. NetVRM further improves *Equal-Active* with the network-wide memory allocation algorithm in §5.

Performance metric. We use *satisfaction ratio* as the performance metric for these network applications. Each application instance has a utility target. The satisfaction ratio of an instance is the fraction of time the utility target is met during its lifetime. For each experiment, we compute the satisfaction ratio for every instance, and show the mean and 5th percentile of the satisfaction ratios across all instances. Considering the number of instances is only a few hundreds (i.e., 256), the 5th percentile catches the tail pattern in the last ten instances, while other options (e.g., 1th, 0.1th) are too limited which only show the satisfaction of the last one or two instances.

8.2.1 Generality

We show that NetVRM is general to a wide range of network application types in the WAN scenario.

Setup. We replay the CAIDA traffic on the four emulated switches as in §8.1. We deploy four types of applications including HH, NO, SS and SHH. We omit NetCache as it is not a good use case for the WAN scenario. HH maintains the flow records of the source IP and the corresponding number of packets for all the IP traffic. NO maintains the flow records of the source IP and the corresponding number of packets only for TCP SYN packets. SS records the distinct IP address
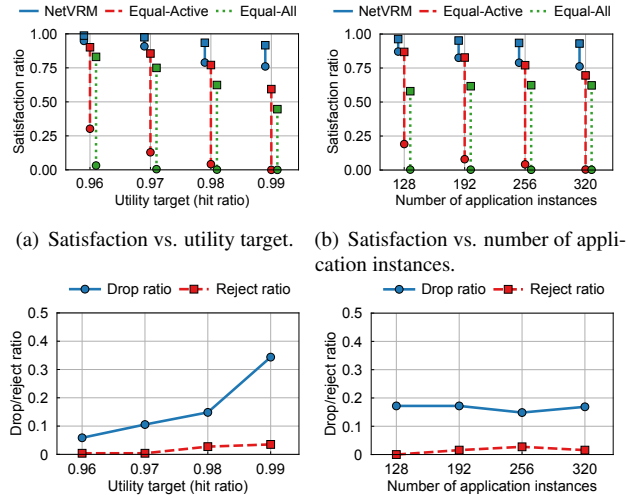
(a) Satisfaction vs. utility target.   (b) Satisfaction vs. number of application instances.

(c) Drop/reject ratio vs. utility target. (d) Drop/reject ratio vs. number of application instances.

Figure 13: Impact of workload parameters.

pair (source IP and destination IP) for all the IP traffic. SHH maintains the flow records of the source IP and the threshold to be identified as a heavy hitter is set to 200. We do the following extension for a network-wide SHH: one SHH's utility is defined as the smallest worst-case accuracy across its switches. Since each stage only supports 32-bit read and write from register memory on the data plane, each virtual slot of the three applications spans two physical stages and there are up to 256K virtual slots (i.e., 2048 KB register memory) on each switch.

By default, there are 256 application instances started in 20 minutes based on a Poisson Process and the running time of the instances follows a uniform distribution from 6 minutes to 14 minutes. The utility targets are specified by the operator based on operational requirements. The default utility target for HH, NO, SS, i.e., the memory hit ratio, is 0.98, and the default utility target for SHH, i.e., the worst-case accuracy, is 0.98. On each switch, we use a /8 instance filter and a /2 switch filter to identify the traffic to be processed by each instance. We feed the CAIDA traces into four switches simultaneously and measure the mean and 5th percentile of satisfaction across the 256 instances.

We remark that *this is only one setup of a demanding workload to stress the system*, following the similar workload pattern in [36, 37]. We show that NetVRM outperforms the alternatives with different workload parameters in §8.2.2.

Results. Figure 10 shows the satisfaction ratios for flow-based applications (i.e., HH, NO, SS) under different amounts of register memory. For each vertical line, the upper square end is the mean satisfaction ratio, and the lower round end is the 5th percentile satisfaction ratio, among the 256 application instances. Figure 10(a), (b) and (c) show the cases that the instances are from the same application type, and (d) shows the case that the instances are from all the three types.
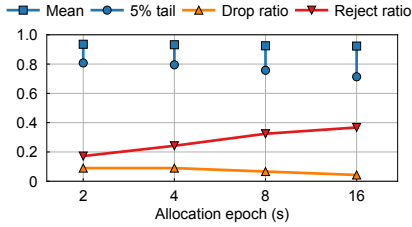
Figure 14: Impact of allocation epochs on NetVRM.

(a) Satisfaction.

(b) Drop/reject ratio.

Figure 15: Experimental results in the datacenter scenario.

When the register memory is limited (e.g., 256 KB), NetVRM significantly outperforms *Equal-All* and *Equal-Active* on both the mean and the tail. When the register memory is abundant (e.g., 2048 KB), NetVRM is able to maintain both high mean and 5th percentile satisfaction ratios. In contrast, *Equal-All* and *Equal-Active* have comparable mean satisfaction ratios, but suffer from the tail behavior. The advantage of NetVRM over *Equal-All* and *Equal-Active* is consistent across different application types. SS uses src IP and dst IP as the hash key. Thus, it has fewer hash collisions than HH and NO, leading to a higher satisfaction ratio. Figure 11 shows the drop ratios and rejection ratios of NetVRM under the four workloads. Similarly, SS drops and rejects fewer application instances than HH and NO, because it has fewer hash collisions and less memory requirement.

Figure 12 shows that NetVRM outperforms *Equal-All* and *Equal-Active* with the sketch-based applications (i.e., SHH) as well. Compared with flow-based applications, the alternatives have lower satisfaction ratios and NetVRM drops more application instances because SHH needs more memory to guarantee the worst-case accuracy bounds.

The alternatives, *Equal-All* and *Equal-Active*, have close performance for all the applications, which means only having the mechanism of virtual register memory to allocate resources to active applications is not sufficient. The allocation algorithm that decides the memory allocation plan is critical to the performance.

8.2.2 Analysis of NetVRM

We analyze NetVRM by showing the impact of workload parameters and the allocation epoch. We use the same setup in §8.2.1 and show the results for the workload of HH. The findings for other application types are similar. We demonstrate the benefits of NetVRM over the local memory allocation approach in Appendix B.

Impact of workload parameters. Figure 13(a) shows that NetVRM is able to manage the register memory efficiently with different utility targets. With more strict targets, the three approaches have worse performance as the application instances have higher memory requirements. Figure 13(c) shows the drop ratio and reject ratio increase with more strict targets. Figure 13(b) studies the impact of the number of application instances arriving in each experiment. Fewer instances mean less resource contention, leading to higher satisfaction. NetVRM consistently outperforms the alternatives. Interestingly, Figure 13(d) shows that the drop ratio and
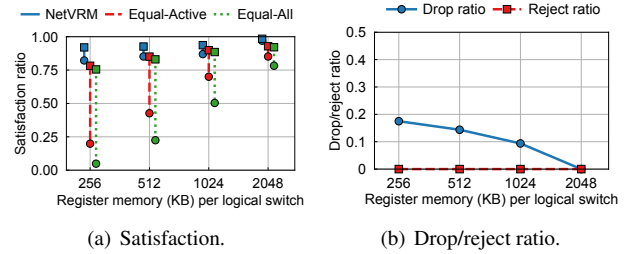
reject ratio are not significantly influenced by the number of instances in the evaluated range.

Impact of the allocation epoch. Figure 14 shows that a shorter allocation epoch leads to a slightly better performance, both in mean and tail. A longer allocation epoch can get a comparable satisfaction ratio but it comes with rejecting more applications. For example, when the allocation epoch is 16 seconds, NetVRM drops and rejects about 40% application instances, while the sum of drop ratio and reject ratio is 25% when the allocation epoch is 2 seconds.

8.2.3 NetVRM in Datacenter Network

Setup. We use the four independent pipelines of the Tofino switch to emulate four switches, and wire the four switches to build a datacenter network topology (shown in Appendix C). S0, S1 and S2 are ToR switches for client rack 1, client rack 2 and the key-value rack respectively. S3 is a spine switch connecting to them. We run two types of applications, which are HH and NetCache. HH records the number of packets of distinct four tuples (source IP, destination IP, source port, destination port). We use the Cluster-C traffic trace from Facebook's production datacenters [41]. The trace is anonymized by hashing. The IP addresses are hashed to 64 bits and the port numbers are hashed to 32 bits in the trace. The HH application uses six physical stages to store the four tuples and one extra stage to store the number of packets. We generate pcap files from the Facebook trace, and assign the timestamps of the packets uniformly in one second as the original timestamp is at second granularity. Each application instance owns a /8 subnet. There are 318 HH instances arriving in 20 minutes based on a Poisson process, and the running time of the instances follows a uniform distribution from 6 minutes to 14 minutes. The HH instances use two paths, S0-S3-S2 and S1-S3-S2. The utility target of HH is set to 0.96.

We run two NetCache instances. NetCache1 (NC1) uses path S0-S3-S2, and NetCache2 (NC2) uses path S1-S3-S2. The tenants of NC1 and NC2 are in client rack 1 and client rack 2, respectively, which access different key-value items in the key-value rack, so they cannot share the memory on S2 and S3. NC1 and NC2 run throughout the 30-minute experiment time. The workload skewness changes between Zipf-0.99 and Zipf-0.95 every 6 minutes. The utility target is 0.5. Each virtual slot of NetCache spans 8 physical stages,

resulting in up to 64K virtual slots per switch. The NetCache instances are set to not be dropped.

Results. Figure 15(a) shows the satisfaction ratios of the three approaches, and Figure 15(b) shows the drop ratios and reject ratios of NetVRM. Similarly, NetVRM outperforms *Equal-All* and *Equal-Active* consistently under different amounts of register memory. It indicates that NetVRM can multiplex the register memory between different switches in a complicated scenario where applications have multiple paths and measurement applications run along with datacenter-specific applications such as NetCache.

9 Conclusion

We present NetVRM, a network management system to support dynamic register memory sharing between multiple concurrent applications on a programmable network. NetVRM provides a *virtual register memory abstraction* that enables register memory sharing in the switch data plane, and dynamically allocates memory for better resource efficiency and application utility. NetVRM also provides P4VRM as an extension of P4 for developing applications with virtual register memory, and a compiler to generate data plane programs and control plane APIs.

References

[1] In-band Network Telemetry (INT) Dataplane Specification. `https://github.com/p4lang/p4-applications/blob/master/docs/INT.pdf`.

[2] P4-14 Language Specification. `https://p4.org/p4-spec/p4-14/v1.0.5/tex/p4.pdf`.

[3] P4 Behavioral Model Repository. `https://github.com/p4lang/behavioral-model`.

[4] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS*, June 2012.

[5] Networking and Content Delivery on AWS. `https://aws.amazon.com/products/networking/`.

[6] Multitenant SaaS on Azure. `https://docs.microsoft.com/en-us/azure/architecture/example-scenario/multi-saas/multitenant-saas`.

[7] Azure networking services overview. `https://docs.microsoft.com/en-us/azure/networking/fundamentals/networking-overview`.

[8] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *SIGCOMM CCR*, July 2014.

[9] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *ACM SIGCOMM*, August 2013.

[10] The CAIDA Anonymized Internet Traces 2019 Dataset. `https://data.caida.org/datasets/passive-2019/`.

[11] Cavium XPliant. `https://www.cavium.com/`.

[12] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 2005.

[13] H. T. Dang, M. Canini, F. Pedone, and R. Soulé. Paxos made switch-y. *SIGCOMM CCR*, April 2016.

[14] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé. NetPaxos: Consensus at network speed. In *ACM SOSR*, June 2015.

[15] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle. Moongen: A scriptable high-speed packet generator. In *ACM SIGCOMM Conference on Internet Measurement Conference*, 2015.

[16] S. K. Fayaz, Y. Tobioka, V. Sekar, and M. Bailey. Bohatei: Flexible and elastic ddos defense. In {*USENIX*} *Security*, 2015.

[17] A. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. Participatory networking: An API for application control of SDNs. In *ACM SIGCOMM*, August 2013.

[18] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger. Sonata: Query-driven streaming network telemetry. In *ACM SIGCOMM*, 2018.

[19] D. Hancock and J. Van der Merwe. Hyper4: Using p4 to virtualize the programmable data plane. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, 2016.

[20] Q. Huang, P. P. Lee, and Y. Bao. Sketchlearn: Relieving user burdens in approximate measurement with automated statistical inference. In *ACM SIGCOMM*, 2018.

[21] X. Jin, J. Gossels, J. Rexford, and D. Walker. CoVisor: A compositional hypervisor for software-defined networks. In *USENIX NSDI*, May 2015.

[22] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica. NetChain: Scale-free sub-RTT coordination. In *USENIX NSDI*, April 2018.

[23] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. NetCache: Balancing key-value stores with fast in-network caching. In *ACM SOSP*, October 2017.

[24] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer. Dynamic scheduling of network updates. In *ACM SIGCOMM*, August 2014.

[25] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford. Hula: Scalable load balancing using programmable data planes. In *ACM SOSR*, March 2016.

[26] F. Kelly and T. Voice. Stability of end-to-end algorithms for joint routing and rate control. *SIGCOMM CCR*, 2005.

[27] D. Kim, Z. Liu, Y. Zhu, C. Kim, J. Lee, V. Sekar, and S. Seshan. Tea: Enabling state-intensive network functions on programmable switches. In *ACM SIGCOMM*, 2020.

[28] D. Kim, J. Nelson, D. R. Ports, V. Sekar, and S. Seshan. Redplane: enabling fault-tolerant stateful in-switch applications. In *ACM SIGCOMM*, 2021.

[29] D. Kim, Y. Zhu, C. Kim, J. Lee, and S. Seshan. Generic external memory for switch data planes. In *ACM HotNets Workshop*, 2018.

[30] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, N. Gude, P. Ingram, et al. Network virtualization in multi-tenant datacenters. In *USENIX NSDI*, April 2014.

[31] J. Levine. *Flex & Bison: Text Processing Tools.* " O'Reilly Media, Inc.", 2009.

[32] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya. IncBricks: Toward in-network computation with an in-network cache. In *ACM ASPLOS*, April 2017.

[33] Z. Liu, Z. Bai, Z. Liu, X. Li, C. Kim, V. Braverman, X. Jin, and I. Stoica. Distcache: Provable load balancing for large-scale storage systems with distributed caching. In *USENIX FAST*, 2019.

[34] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *ACM SIGCOMM*, 2016.

[35] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *ACM SIGCOMM*, 2017.

[36] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. DREAM: Dynamic resource allocation for software-defined measurement. In *ACM SIGCOMM*, August 2014.

[37] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. Scream: Sketch resource allocation for software-defined measurement. In *ACM CoNEXT*, 2015.

[38] K. Nagaraj, D. Bharadia, H. Mao, S. Chinchali, M. Alizadeh, and S. Katti. Numfabric: Fast and flexible bandwidth allocation in datacenters. In *ACM SIGCOMM*, 2016.

[39] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim. Language-directed hardware design for network performance monitoring. In *ACM SIGCOMM*, August 2017.

[40] V. Nathan, V. Sivaraman, R. Addanki, M. Khani, P. Goyal, and M. Alizadeh. End-to-end transport for video qoe fairness. In *ACM SIGCOMM*, 2019.

[41] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *ACM SIGCOMM*, 2015.

[42] A. Sapio, I. Abdelaziz, M. Canini, and P. Kalnis. Daiet: a system for data aggregation inside the network. In *ACM Symposium on Cloud Computing*, 2017.

[43] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. R. K. Ports, and P. Richtárik. Scaling distributed machine learning with in-network aggregation, 2019.

[44] M. Saquetti, G. Bueno, W. Cordeiro, and J. R. Azambuja. P4vbox: Enabling p4-based switch virtualization. *IEEE Communications Letters*, 2019.

[45] N. K. Sharma, A. Kaufmann, T. E. Anderson, A. Krishnamurthy, J. Nelson, and S. Peter. Evaluating the power of flexible packet processing for network resource allocation. In *USENIX NSDI*, March 2017.

[46] R. Sherwood, G. Gibb, K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Can the production network be the testbed? In *USENIX OSDI*, October 2010.

[47] J. Sonchack, A. J. Aviv, E. Keller, and J. M. Smith. Turboflow: Information rich flow record generation on commodity switches. In *EuroSys*, 2018.

[48] H. Soni, T. Turletti, and W. Dabbous. P4Bricks: Enabling multiprocessing using linker-based network data plane architecture. 2018.

[49] R. Stoyanov and N. Zilberman. Mtpsa: Multi-tenant programmable switches. In *Proceedings of the 3rd P4 Workshop in Europe*, 2020.

[50] Intel Tofino. `https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html`.

[51] T. Wang, H. Zhu, F. Ruffy, X. Jin, A. Sivaraman, D. R. K. Ports, and A. Panda. Multitenancy for fast and programmable networks in the cloud. In *USENIX HotCloud Workshop*, 2020.

[52] D. Wu, A. Chen, T. E. Ng, G. Wang, and H. Wang. Accelerated service chaining on a single switch asic. In *ACM HotNets Workshop*, 2019.

[53] L. Yu, J. Sonchack, and V. Liu. Mantis: Reactive programmable switches. In *ACM SIGCOMM*, August 2020.

[54] M. Yu, L. Jose, and R. Miao. Software defined traffic measurement with opensketch. In *USENIX NSDI*, 2013.

[55] Y. Yuan, D. Lin, A. Mishra, S. Marwaha, R. Alur, and B. T. Loo. Quantitative network monitoring with netqre. In *ACM SIGCOMM*, 2017.

[56] C. Zhang, J. Bi, Y. Zhou, A. B. Dogar, and J. Wu. Hyperv: A high performance hypervisor for virtualization of the programmable data plane. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, 2017.

[57] P. Zheng, T. Benson, and C. Hu. P4visor: Lightweight virtualization and composition primitives for building and testing modular programs. In *ACM CoNEXT*, 2018.

[58] H. Zhu, Z. Bai, J. Li, E. Michael, D. Ports, I. Stoica, and X. Jin. Harmonia: Near-linear scalability for replicated storage with in-network conflict detection. In *Proceedings of the VLDB Endowment*, November 2019.

A Diminishing Return Examples

Figure 16 demonstrates the diminishing returns for four applications. The first three are measurement applications: heavy hitter detection (HH) [54], newly opened TCP connection detection (NO) [55], superspreader detection (SS) [54]. These applications store flow records in the data plane; hash collisions caused by inadequate memory require additional control plane processing. The fourth, NetCache [23] caches hot objects in the switch data plane to improve the throughput of a key-value store. The utility is measured using memory hit ratio. We evaluate the measurement applications (Figure 16(a–c)) on traffic from different subnets of the 2019 passive CAIDA trace [10], and NetCache on a synthetic Zipf workload with different skewness parameters (Figure 16(d)).

(a) Heavy hitter detection.   (b) Newly opened TCP connections.

(c) Superspreader detection.   (d) NetCache.
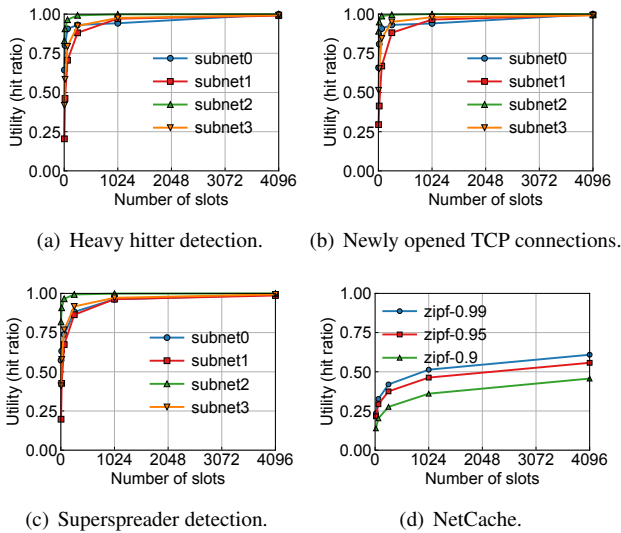
Figure 16: Examples for the diminishing returns of the utility curves in reg-stateful network applications.

B Additional Evaluation Results

(a) Satisfaction.   (b) Drop/reject ratios of NetVRM and *Local-Alloc*.
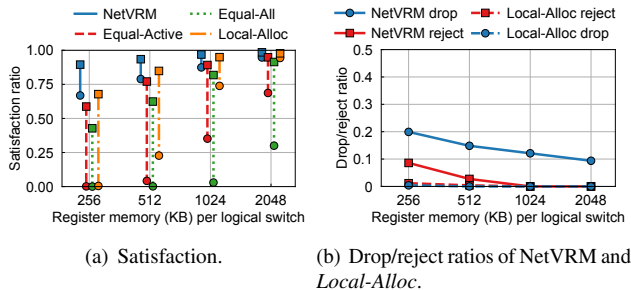
Figure 17: Comparison with *Local-Alloc*.

Comparison with local memory allocation. Besides the *Equal-all* and *Equal-Active*, we also compare NetVRM with *Local-Alloc* which only does memory allocation and makes drop/reject decisions on individual switches locally. One application is counted as drop/reject only after all the four switches have decided to drop/reject it. We report the results for HH workload. The findings for other application types are similar. Figure 17 shows that *Local-Alloc* has better performance than *Equal-all* and *Equal-Active*, but is still worse than NetVRM because it fails to capture network-wide information and makes sub-optimal allocation and drop/reject decisions.

C Network Topology in Datacenter Scenario

We wire the four emulated switches to build a datacenter network topology, as shown in Figure 18, to evaluate the performance of NetVRM in the datacenter scenario.
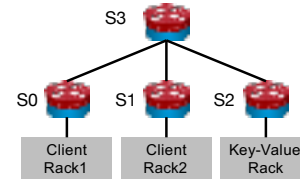
Figure 18: Datacenter topology for evaluation.

SwiSh: Distributed Shared State Abstractions for Programmable Switches

Lior Zeno[⋆]   Dan R. K. Ports[†]   Jacob Nelson[†]   Daehyeok Kim[†]   Shir Landau Feibish[‡]   Idit Keidar[⋆]

Arik Rinberg[⋆]   Alon Rashelbach[⋆]   Igor De-Paula[⋆]   Mark Silberstein[⋆]

[⋆]Technion   [†]Microsoft Research   [‡]The Open University of Israel

Abstract

We design and evaluate *SwiSh*, a *distributed shared state management* layer for data-plane P4 programs. SwiSh enables running scalable stateful distributed network functions on programmable switches entirely in the data-plane. We explore several schemes to build a shared variable abstraction, which differ in consistency, performance, and in-switch implementation complexity. We introduce the novel *Strong Delayed-Writes (SDW)* protocol which offers consistent snapshots of shared data-plane objects with semantics known as *r*-relaxed strong linearizability, enabling implementation of distributed concurrent sketches with precise error bounds.

We implement strong, eventual, and SDW consistency protocols in Tofino switches, and compare their performance in microbenchmarks and three realistic network functions, NAT, DDoS detector, and rate limiter. Our results show that the distributed state management in the data plane is practical, and outperforms centralized solutions by up to four orders of magnitude in update throughput and replication latency.

1 Introduction

In recent years, programmable data-plane switches such as Intel's Tofino, Broadcom's Trident, and NVIDIA's Spectrum [9, 33, 60] have emerged as a powerful platform for packet processing, capable of running complex user-defined functionality at Tbps rates. Recent research has shown that these switches can run sophisticated network functions (NFs) that power modern cloud networks, such as NATs, load balancers [40, 57], and DDoS detectors [45]. Such data-plane implementations show great promise for cloud operators, as programmable switches can operate at orders of magnitude higher throughput levels than the server-based implementations used today, enabling a massive efficiency improvement.

A key challenge remains largely unaddressed: realistic data center deployments require NFs to be *distributed over multiple switches*. Multi-switch execution is essential to correctly process traffic that passes through multiple network paths,

to tolerate switch failures, and to handle higher throughput. Yet, building distributed NFs for programmable switches is challenging because most of today's NFs are *stateful* and need their state to be consistent and reliable. For example, a DDoS detector may need to monitor traffic coming from multiple locations via several switches. However, it cannot be implemented by routing all traffic through a single switch since it is inherently not scalable. Instead, it must be implemented in a distributed manner. Furthermore, in order to detect and mitigate an attack, a DDoS detector must aggregate per-packet source statistics across all switches in order to correctly identify super-spreaders sending to too many destinations. Similarly, in multi-tenant clouds, per-user policies, such as rate limiting, cannot be implemented in a single switch because user's VMs are often scattered across multiple racks, so the inter-VM traffic passes through multiple switches.

Distributed state management is, in general, a hard problem, and it becomes even harder in the context of programmable data-plane switches. In the "traditional" host-based NF realm, several methods have been proposed to deal with distributed state. These include remote access to centralized state storage [39] and distributed object abstractions [77], along with checkpoints and replication mechanisms for fault tolerance [64, 71]. Unfortunately, few of these techniques transfer directly to the programmable switch environment. These switches have the capability to modify state on *every* packet, allowing them to effectively implement stateful NFs. However, distributing the NF logic across multiple switches is extremely challenging as it requires synchronizing these frequent changes under harsh restrictions on computation, memory and communication.

Existing systems that implement NFs over multiple switches do so by designing ad hoc, application-specific protocols. Recent work on data-plane defense against link flooding [36], argues for data-plane state synchronization among the switches, but provides no consistency guarantees. While applicable in this scenario, it would not be enough in other applications, as we discuss in our analysis (§4). A more common solution, usually applied in network telemetry systems,

is to periodically report the per-switch state to a central controller [1,6,18,25,26,29,49,78]. Such systems need to manage the state kept on each switch and to determine when and how the central controller is updated – navigating complex tradeoffs between frequent updates leading to controller load and communication overhead versus stale data leading to measurement error. In contrast to these approaches, we seek a solution that supports general application scenarios *without relying on a central controller in failure-free runs*, while allowing all switches to take a *consistent action as a function of the global state*, e.g., to block a suspicious source in the DDoS detector.

We describe the design of such a general distributed shared state mechanism for data-plane programs, **SwiSh**. Inspired by distributed shared memory abstractions for distributed systems [41, 48], SwiSh provides several replicated shared variable abstractions with different consistency guarantees. At the same time, SwiSh is tailored to the needs of NFs and co-designed to work in a constrained programmable switch environment.

Our analysis reveals three families of NFs that lend themselves to efficient in-switch implementation, with distinct consistency requirements. For each family we explore the triple tradeoff between consistency, performance, and complexity. We design (1) Strong Read-Optimized (SRO): a strongly consistent variable for read-intensive applications with low update rates, (2) Eventual Write-Optimized (EWO): an eventually-consistent variable for applications that can tolerate inconsistent reads but require frequent writes, and (3) Strong Delayed-Writes (SDW): a novel consistency protocol which efficiently synchronizes multi-variable snapshots across switches while providing a consistency and correctness guarantee known as r-relaxed strong linearizability [27].

SDW is ideal for implementing concurrent sketches, which are popular in data-plane programs [12, 13, 24, 30, 35, 38, 51–54, 78, 83]. Unlike eventually consistent semantics, the r-relaxed strong linearizability offered by SDW enables principled analysis of concurrent sketches. This property enables the derivation of precise error bounds and generalizes to different sketch types, such as non-commutative sketches [67].

Implementing these abstractions efficiently in a switch is a challenge, and it involves judicious choice of hardware mechanisms and optimization targets. Our main ideas are: (1) *minimizing the buffer space* due to the scarcity of switch memory, even at the expense of higher bandwidth; (2) using the *in-switch packet generator* for implementing reliable packet delivery and synchronization in the data-plane.

We fully implement all the protocols in Tofino switches and devise reusable APIs for data-plane replication. We evaluate the protocols both in micro-benchmarks and in three real-world distributed NFs: a rate limiter, a network address translator (NAT) and a DDoS detector. Our novel SDW protocol achieves micro-second synchronization latency and offers about four orders of magnitude higher update rates compared to a central controller or SRO. We show that SDW (1) achieves stable 99th percentile replication latency of 6μsec when running on four programmable pipes (two per switch), thus sharing state both among local and remote pipes; (2) scales to 32 switches when executed in a large-scale emulation and fits switch resources even for 4K switches; (3) requires linear number of replication messages in state size which is independent from the number of actual updates to the state.

We show that SDW is instrumental to achieving high performance in applications: the centralized controller fails to scale under growing application load, whereas SDW-based versions show no signs of performance degradation.

This paper extends our workshop paper [82] by introducing the SDW protocol, as well as providing an implementation and evaluation of SRO and EWO.

In summary, this work makes the following contributions:
- Analysis of memory consistency requirements and access patterns of common NFs suitable for in-switch execution,
- Design and implementation of strongly- and eventually consistent shared variables, as well as a new SDW consistency protocol specifically tailored for in-switch implementation, which guarantees consistent snapshots and provably provides r-relaxed strong linearizability which facilitates implementation of concurrent sketches,
- An implementation and evaluation of three distributed NFs on Tofino switches demonstrating the practicality and utility of the new abstractions.

2 Background: Programmable Switches

The protocol independent switch architecture (PISA) [8] defines two main parts to packet processing. The first is the parser which parses relevant packet headers, and the second is a pipeline of match-and-action stages. Parsed headers and metadata are then processed by the pipeline. The small (∼10 MB) switch memory is shared by all pipeline stages. Often, switches are divided into multiple independent pipes [34], each serving a subset of switch ports. From the perspective of in-switch applications, the pipes appear as different switches, so stateful objects are not shared between them.

PISA-compliant devices can be programmed using the P4 language [73]. P4 defines a set of high-level objects that consume switch memory: tables, registers, meters, and counters. While tables updates require control-plane involvement, all other objects can be modified directly from the data-plane.

A data-plane program processes packets, and then can send them to remote destinations to the control-plane processor on the switch, or to the switch itself (called *recirculation*).

Switches process packets atomically: a packet may generate several local writes to different locations, and these updates are atomic in the sense that the next processed packet will not see partial updates. Single-row control-plane table updates are atomic w.r.t. data-plane [74]. These properties allow us to implement complex distributed protocols with concurrent state updates without locks.

Although not a part of PISA, some switches add packet generation support. Packet generators can generate packets directly into the data-plane. For example, the Tofino Native Architecture (TNA) [34] allows generation of up to 8 streams of packets based on templates in switch memory. The packet generator can be triggered by a timer or by matching certain keys in recirculated packets.

3 Motivation

The Case for Programmable Switches as NF Processors. The modern data center network incorporates a diverse array of NFs beyond simple packet forwarding. Features like NAT, firewalls, load balancers, and intrusion detection systems are central to the functionality of today's cloud platforms. These functions are stateful packet processing operations, and today are generally implemented using software middleboxes that run on commodity servers, often at significant cost.

Consider an incoming connection to a data center service. It may pass through a DDoS detection NF [3, 58], which blocks suspicious patterns. This service is stateful; it collects global traffic statistics, e.g., to identify "super-spreader" IPs that attempt to flood multiple targets. Subsequently, traffic may pass through a load balancer, which routes incoming TCP connections to multiple destination hosts. These are stateful too: because subsequent packets in the same TCP connection must be routed to the same server (a property dubbed per-connection consistency), the load balancer must track the connection-to-server mapping. Both DDoS detectors and load balancers are in use at major cloud providers [19, 61], and handle a significant fraction of a data center's incoming traffic. Implemented on commodity servers, they require large clusters to support massive workload.

Programmable data-plane switches offer an appealing alternative to commodity servers for implementing NFs at lower cost. Researchers have shown that they can be used to implement many types of NFs. For data center operators, the benefit is a major reduction in the cost of NF processing. Whereas a software-based load balancer can process approximately 15 million packets per second on a single server [19], a single switch can process 5 *billion* packets per second [33]. Put another way, a programmable switch has a price, energy usage, and physical footprint on par with a single server, but can process *several hundred times* as many packets.

Distributed Switch Deployments. Prior research focused on showing that NFs can be implemented on a single switch [45, 57]. However, realistic data center deployments universally require multiple switches. We see two possible deployment scenarios. The NF can be placed in switches in the network fabric. For example, in order to capture all traffic, the load balancer would need to run on all possible paths, e.g., by being deployed on every core switch or every aggregation switch. Alternatively, a cluster of switches (perhaps located near the ingress point) could be used to serve as NF accelerators. Both

are inherently distributed deployments: they require multiple switches in order to (1) scale out, (2) tolerate switch failures, and (3) capture traffic across multiple paths.

The challenge of a distributed NF deployment stems from the need to manage the global state shared among the NF instances, which is inherent to distributed stateful applications. Specifically, packet processing at one switch may require reading or updating variables that are also accessed by other switches. For example, the connection-to-server mapping recorded by the load balancer must be available when later packets for that connection are processed – even if they are processed by a different switch, or the original switch fails. Similarly, a rate limiter would need to track and record the total incoming traffic from a given IP, regardless of which switch is processing it.

SwiSh provides a shared state mechanism capable of supporting global state: any global variable can be read or written from any switch. SwiSh transparently replicates state updates to other switches for fault tolerance and remote access. In case of state locality, only a subset of the switches would replicate that state [82].

The Case for Data-Plane Replication. Control-plane mechanisms are commonly used for replicating the switch state [7, 11, 43, 56]. However, the scalability limitations of this approach have been well recognized, and several recent works focus on improving it by distributing the control-plane logic across a cluster of machines or switches [43, 81]. SwiSh proposes instead to replicate the state in the data plane.

Data plane replication enables supporting distributed NFs that read or modify switch state on *every packet*. This new capability of programmable data-plane switches allows implementations of more sophisticated data-plane logic than traditional control-plane SDN.

As we will see in §4, applications use state in diverse ways. Some are read-mostly; others update state on every packet. Some require strong consistency among switches to avoid exposing inconsistent states to applications (e.g., a distributed NAT must maintain correct mappings to avoid packet loss), while others can tolerate weak consistency (e.g., rate limiters that already provide approximate results [63]). SwiSh provides replication mechanisms for different classes of data that operate at the speed of the switch data-plane.

At the same time, data-plane replication offers an opportunity to build a more efficient replication mechanism without additional control-plane processing servers. Furthermore, data-plane replication can take advantage of unique programmable hardware characteristics that are not available in a traditional control-plane. For example, the atomic packet processing property enables a multi-location atomic write to the shared state. We leverage this feature to enable fast processing of acknowledgments entirely in the data-plane for our strongly-consistent replication protocol (§6.1).

Control-plane replication is not enough. Managing a globally shared state in a programmable data-plane switch requires
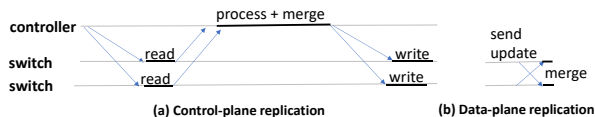
Figure 1: Data-plane vs. control-plane replication

a new approach: replication protocols that run in the control-plane cannot operate at this rate at scale.

Figure 1 shows the cycle performed by a controller to synchronize between switches, and contrasts that with data-plane replication. The controller periodically queries the switches, collects information, processes it, and sends the updates back. Merely reading and updating the register states in switches is quite slow. We measured an average latency of 507msec to read a sketch with 3 rows each with 64K 4-byte registers from the on-switch control-plane;[1] updates are similar. This latency limits the rate at which the data can be retrieved from switches.

Moreover, the central controller may become the bottleneck quite quickly. For example, recent work on DDoS detection that used a central controller to query switches reported a maximum update rate of once in 5 seconds [53] because it could not accommodate faster updates.

In contrast, data-plane replication reads from and writes to registers much faster: we measured 486μseconds to read the same sketch from the data-plane, which is over three orders-of-magnitude faster than the control-plane access. Further, in-switch processing time is negligible as well.

These properties make data-plane replication an obvious choice for building stateful distributed NFs.

4   Application Consistency Requirements

We study the access patterns and consistency requirements of a few typical NF applications that have been built on PISA switches. Table 1 summarizes the results.

We identify three families of consistency requirements:

1. **Strong consistency**: Workloads cannot tolerate inconsistency between switches – a read must see a previous write. These are usually read-intensive workloads that can tolerate infrequent, but expensive writes;

2. **Weak (eventual) consistency**: Mixed read/write workloads tolerate arbitrary inconsistency;

3. **Bounded-delay consistent snapshots**: Mixed read/write workloads that tolerate inconsistency for a bounded time – a read must see all but a bounded number of previous writes, yet require that *all switches* read from a consistent state. These requirements are typical for sketches.

Below, we describe how these consistency requirements arise in several in-switch applications.

4.1   Strong Consistency

Network Address Translators (NATs) share the connection table among the NF instances. The table is queried on every packet, but updated when a new connection is opened; table rows require strong consistency, or it may lead to broken client connections in case of multi-path routing or switch failure. Also, NATs usually manage a pool of unassigned ports; however, the pool can be partitioned among the switches into non-overlapping ranges to avoid sharing.

Stateful firewalls monitor connection states to enforce context-based rules. These states are stored in a shared table, updated as connections are opened and closed, and accessed for each packet to make filtering decisions. Like the NAT, the firewall NF requires strong consistency to avoid incorrect forwarding behavior.

L4 load balancers [57] assign incoming connections to a particular destination IP, then forward subsequent packets to the appropriate destination IP. Per-connection consistency requires that once an IP is assigned to a connection, it does not change, implying a need for strong state consistency.

Observation 1. These workloads require strong consistency, but they update state infrequently, making a costly replication protocol more tolerable. Moreover, most of these examples use switch tables that should be modified *through the control-plane*, naturally limiting their update rate. For example, the NAT NF uses control-plane to update the connection table. We leverage this observation when designing the replication protocol for this class of NFs.

4.2   Weak (Eventual) Consistency

Rate limiters restrict the aggregated bandwidth of flows that belong to a given user. The application maintains a per-user meter that is updated on every packet. The meters are synchronized periodically to identify users exceeding their bandwidth limit and to enforce restrictions. Maintaining an exact network-wide rate across all switches would incur a very high overhead and is therefore unrealistic. So rate limiters can tolerate inconsistencies, but the meters must be synchronized often enough [63] to minimize discrepancy.

Intrusion prevention systems (IPS) [47] monitor traffic by continuously computing packet signatures and matching against known suspicious signatures. If the number of matches is above a threshold, traffic is dropped to prevent the intrusion. This application can tolerate transient inconsistencies: it is acceptable for a few malicious packets to go through immediately after signatures are updated.

Observation 2. Some NFs tolerate weakly consistent data, potentially affording simpler and more efficient replication protocols. However, as we will describe next, other functions may defer the writes to be once in a window, but do require to have a consistent view of prior writes among all the switches.

---

[1] We use BfRt API (C++) and average over 100 iterations.

	Application	State	Write frequency	Read frequency
Strong consistency	NAT	Translation table	New connection	Every packet
	Firewall	Connection states table	New connection	Every packet
	L4 load-balancer	Connection-to-DIP mapping	New connection	Every packet
Weak consistency	Intrusion prevention system	Signatures	Low	Every packet
	Rate limiter	Per-user meter	Every packet	Every window
Bounded delay consistent snapshot	DDoS detection	Sketch	Every sampled packet	Every packet
	Microburst detection	Sketch	Every packet	Every window

Table 1: NFs classified by their access pattern to shared data and their consistency requirements.

4.3 Bounded-Delay Consistent Snapshots

We assign mixed read/write applications that use data sketches to this class. Data sketches are commonly used in data-plane programs [12, 13, 24, 30, 35, 51, 52, 54, 78]. They are probabilistic data structures that efficiently collect approximate statistics about elements of a data stream.

Below we consider two examples of sketch-based NFs.

Microburst detection identifies flows that send a lot of data in a short time period. ConQuest [13] is a recent sketch-based system for a single switch, which uses a sliding window mechanism composed of a group of Count-Min sketches (CMS) [14]. At most one sketch is updated on every packet.

DDoS detection [45] requires tracking the frequency of source and destination IPs using a CMS with bitsets [80]. The sketch is updated on every packet, but sampled periodically to trigger an alarm when IP frequencies cross a threshold.

Strongly consistent read-optimized protocols are too costly for such workloads due to their write-intensive nature. Fortunately, because a data sketch is inherently approximate, it does not require strong consistency – it is acceptable for a query to miss some updates. Moreover, sketches are typically *stream-order invariant* [67], meaning that the quantity they estimate (such as number of unique sources, heavy hitters, and quantiles) does not depend on the packet order.

At the same time, sketches generally cannot tolerate weak consistency either. With no guarantee of timeliness, sketches might be useless. A DDoS attack might be over by the time it is detected. Moreover, the attack might be detected at one location much earlier than it is detected at another, leading to an inconsistent response. Furthermore, sketches have known error bounds (see [15] and others). These bounds are violated if updates are arbitrarily delayed [27, 66], making it hard to reason about the impact of sketch errors on the application.

Observation 3. Sketches require a *bounded-delay consistent snapshot* consistency level. Formally, it provides *r-relaxed strong linearizability* (Appendix A), which supports sketch applications with provably bounded error. Intuitively, *r*-relaxed strong linearizability guarantees that accesses to shared data are equivalent to a sequential execution, except that each query may "miss" up to *r* updates. SwiSh supports this consistency level using its novel Strong Delayed-Write (SDW) protocol,

which provides a consistent snapshot of the sketch at all the replicas, while delaying reads until such a snapshot is constructed.

5 SwiSh Abstractions

SwiSh provides the abstraction of shared variables to programmable switches. This section describes the interface and the types of semantics it offers for shared data.

System model. We consider a system of many switches, each acting as a replica of shared state. Switches communicate via the network, and we assume a standard failure model: packets can be dropped, duplicated and arbitrarily re-ordered, and links and switches may fail. Since switches are comprised of multiple independent pipes with per-pipe state (§2), we consider a pipe rather than a switch, a node in the protocol. We use the terms pipe and switch interchangeably.

Data model. The basic unit of shared state is a *variable*, associated with a unique key, which exposes an API for updating the variable (potentially using general read-modify-write functions), and reading it. The API is thus available on all switches, and variables are read and updated through a distributed protocol. SwiSh supports three types of variables which have different semantics and are accessed through different protocols:

1. *Strong Read-Optimized (SRO)* variables provide strong consistency (linearizability);
2. *Eventual Write-Optimized (EWO)* variables have low cost for both reads and writes, but provide only eventual consistency;
3. *Strong Delayed-Writes (SDW)* variables provide strong consistency (linearizability), but expose writes (even to the local replica) only after their values have been synchronized across the replicas.

We require that, no matter which semantics are used, all variables eventually converge to a common state. To this end, we require that variables be *mergeable*. We consider two merging policies: LWW as a general method, and Conflict-Free Replicated Data Types (CRDTs) as specialized mergeable data types that implement common data structures that are used in NFs. A general way to merge variables is to assign

an order to updates and apply a last-writer-wins (LWW) policy. The merge function applies an update if and only if its version number is larger than the local one. Unique version numbers can be obtained by using a switch ID as a tie breaker in addition to a timestamp attached to each write request.

In some cases, updates can be merged systematically. These are discussed in the literature of Conflict-Free Replicated Data Types (CRDTs), which offer *strong eventual consistency* and *monotonicity* [69]. Monotonicity prevents counter-intuitive scenarios such as an increment-only counter decreasing.

Counters are a natural application for this technique, as they are common in NFs (§4) and have a straightforward CRDT design. An increment-only counter can be implemented by maintaining a *vector* of counter values, one per switch. To update a counter, a switch increments its own element; to read the result, it sums all elements. To merge updates from another switch, a switch takes the largest of the local and received values for each element. Further extensions support decrement operations [69].

Variables may be used to store different data types, such as array entries, read/write variables, sets, and counters. They are implemented using appropriate stateful P4 objects.

6 In-Switch Replication Protocols

Below we assume that switches do not fail; we relax this assumption in §6.4.

6.1 Strong-Read Optimized (SRO)

The SRO protocol is based on chain replication [76], as shown in Figure 2a, adapted to an in-switch implementation with the following key difference: instead of contacting the tail for its latest version and keeping multiple versions per variable, we forward reads to pending writes to the tail.

SRO provides per-variable linearizability [28], because writes are blocking and reads concurrent to writes are processed by the tail node. Its write throughput is limited by the need to send packets through the control plane.[2] Note, however, that many read-intensive NFs already require control plane involvement for their updates, such as NATs, firewall and load balancers [57].

A variation of this protocol, used in many systems, including CRAQ [72] and ZooKeeper [31], reduces the read latency by performing local reads, yet offers weaker semantics [46].

6.2 Eventual Write-Optimized (EWO)

Both variants of the read-optimized protocol have a high write cost. Because supporting both strong consistency and fre-

---

[2]NetChain [37] implements chain replication entirely in the data plane. The difference is that NetChain is a service and clients are responsible for retrying operations. Our switches are effectively the "clients" and must buffer output packets and retry requests.

quent updates is fundamentally challenging, we offer relaxed-consistency variables. This is acceptable for many write-intensive applications, as discussed in §4.

Reads from EWO variables are performed locally, and writes are applied asynchronously. That is, when a switch receives a packet P that modifies state, it modifies its local state, emits any output packet P' immediately, and asynchronously sends a write request to all other switches (Figure 2b). A more sophisticated version can employ batching to avoid flooding the network with updates, and instead send the write request after accumulating several updates.

Unlike SRO, we do not delegate the problem of reliable write delivery to the control plane because it does not scale for write-intensive workloads. Instead, switches periodically synchronize each EWO variable from the data plane. This design choice avoids expensive buffering and re-transmission logic in the data-plane.

Periodic synchronization overcomes the issue of lost packets. As updates to EWO variables are idempotent, packets can be arbitrarily duplicated with no effect. Finally, due to updates being commutative, packet reordering has no effect.

We note that this protocol is simple, but it leads to inconsistent replicas and would incur high bandwidth overheads. With over-subscribed links [23], excessive replication traffic would only worsen the congestion. The following protocol overcomes these limitations.

6.3 Strong Delayed-Writes (SDW)

As explained in §4, certain NFs tolerate inconsistencies among switches, but require state convergence within a bounded time. For such NFs, SwiSh offers *strong delayed-writes* (SDW) variables, ensuring semantics known as *r-relaxed strong linearizability* [27]. These semantics guarantee that every read of a variable observes all but a bounded number of updates. If the variable is used to store a data sketch, then *r*-relaxed strong linearizability often directly implies error bounds on the sketch's estimate [67].

SwiSh batches updates into windows, and synchronizes window advancement (Figure 2c). The complete protocol and its analysis appear in Algorithm 1 in Appendix A; below is an informal overview.

To distribute a variable R, each switch maintains three objects holding copies of R: R_u, R_r and R_s. At any given time, R_u is updated, R_r is queried, and R_s is synchronized (merged) across switches. The objects' roles are switched in a round-robin manner on window advancement.

All switches run the same protocol. At the start of a window, all switches send the contents of R_s to all the others. Any (local) update is applied to R_u, and any query is executed on R_r. Once a switch receives R_s from all other switches, and furthermore receives ACKs from all other switches that they received its R_s, it advances to the next window.
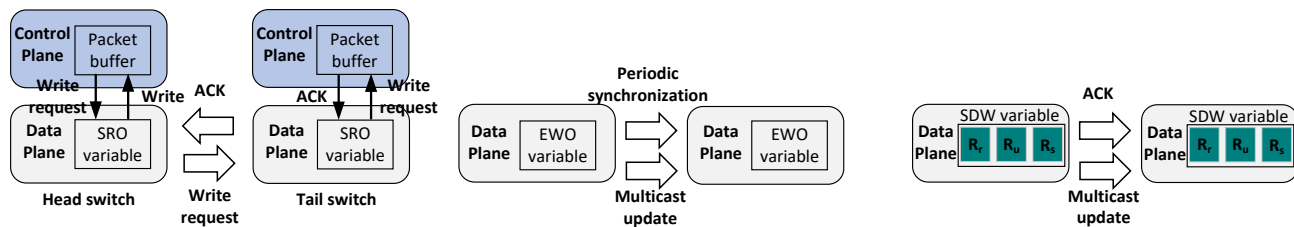
(a) SRO: Based on chain replication. Relies on control-plane for packet buffering.

(b) EWO: Updates are broadcast. Switches periodically send their state for reliability.

(c) SDW: Updates are sent in rounds. Switches advance to the next round after receiving ACKs and updates from others.

Figure 2: A high-level overview of in-switch replication protocols.

On window advancement, the objects are rotated, so R_u becomes the new synchronization variable R'_s, R_r is merged into R_s and then cleared – it becomes the new update object R'_u, and the synchronized buffer R_s becomes the new read buffer R'_r. Thus, after the synchronization of window w completes, R'_u is empty and ready to accumulate updates of window $w+1$, R'_r reflects all updates that occurred in all switches in all windows up to $w-1$, and R'_s reflects all updates done in windows up to $w-1$ in all switches, as well as local updates done in window w.

Crucially, as we prove in Appendix A, this protocol *guarantees* that a query in some window w sees all updates occurring in all windows $\leq w-2$. We also prove that, by bounding the number of updates in a window to B, every query sees all but at most $2NB$ updates that occur before it, where N is the number of switches.

Multi-variable snapshots. Another advantage of the window protocol is that it allows applications to take *consistent snapshots* [59] over a collection of SDW variables by advancing the window simultaneously for all of them. This means that we can support multi-variable queries (for instance, collecting an array of counters as used in a CMS), and ensure that all queries see update batches in a consistent order. Thus, given two updates u_1 and u_2 occurring in different switches, it is impossible for a query at one switch to see a state reflecting only u_1 (and not u_2) while a query at another switch sees only u_2 (and not u_1).

6.4 Handling Failures

We now consider fail-stop switch failures. We assume that a central controller can detect which switches have failed.
SRO. When a switch fails, the chain becomes partitioned. First, we reconnect the chain by bypassing the failed node; if the failed switch is the head, the second node in the chain assumes its responsibility. This follows the standard chain replication protocol. A new switch is added to the end of the chain. It starts to process writes, but does not replace the tail

until the data transfer to it is complete. This requires control plane involvement.

The control plane on one of the switches takes a snapshot of its state, and then resends all pending write requests through the normal data plane protocol. These writes contain the sequence number at the time of the snapshot to prevent overwriting newer values with old ones. Once the new switch has acknowledged all writes, it replaces the tail.
EWO. Because live replicas regularly synchronize their entire state, this synchronization protocol is inherently robust to switch and link failures. The failed switch is removed from the multicast group. Once a new switch replaces the faulty one, it is added to the multicast group, and begins serving reads after obtaining an initial view of the shared state.
SDW. The protocol inevitably stalls once a failure occurs (i.e., the local window ids stop increasing). Denote the maximum window at a correct switch at the time of the failure by *wmax*. The difference between the local window ids at each pair of switches is at most one. Thus, every stalled switch is in window *wmax* or *wmax* − 1.

We reconcile the states of the surviving switches as follows: a controller reads the states of all switches. It collects the state of R_r in some switch that is in window *wmax* and sends it to all switches that are in window *wmax* − 1 (if any), so they advance to window *wmax*. The controller merges all the R_s objects to yield the most up-to-date state for window *wmax* + 1 and broadcasts it to all switches, thus updating their R_s objects to the merged state. Then it removes the failed switch from the multicast group and the switches resume the protocol from window *wmax* + 1.

Adding a new replica is a two-stage process: increasing the expected number of ACKs on correct switches and making sure that all switches are in the same window, which stalls window progression, followed by adding the new replica to the multicast group of each correct switch. The new switch begins serving reads after the current window completes.

We note that during the recovery the updates to the live switches are not lost, but rather accumulated in local switch

replicas R_u. These updates are then synchronized during the recovery. Thus, this protocol is not time critical and can be implemented in control-plane without adding code to the resource-constrained data-plane.

7 Design

We explain the messaging mechanism shared by all protocols, and then describe the SDW design. SRO and EWO closely follow their descriptions in §6.

7.1 Replication Message Exchange

Packet format. Switches exchange replication packets, updates, and acknowledgments with each other to replicate state. Replication packets are IP packets; therefore, by assigning an IP per switch, these packets can be routed using standard L3 routing protocols. Besides Ethernet and IP headers, each packet includes a single bit indicating whether the packet is an update/write request or an ACK, the keys and values accessed by the write, and, in SRO, also a sequence number. For example, in an SRO NAT implementation, the keys are the source IP and source port, and the values are the translated IP and port. In an SDW DDoS application, the keys are sketch indices and the values are counter increments.

Reliable delivery. A major challenge in data-plane replication is ensuring delivery of replication packets. Current switches do not provide enough control over internal switch buffers to store and retransmit a packet from the data-plane.

We identify two cases that require buffering. First, there are *replication* packets generated by each switch as part of the replication protocol. Such packets must be reliably delivered in SDW. Second, there are *write* packets that are received from external sources (not from a switch) and update the NF state in a switch. In SRO these packets cannot be externalized until the updated state is synchronized among the switches.

We handle these two cases separately. For SDW replication packets we keep the state being replicated at the application level until acknowledged, instead of buffering the packet. Then an *ACK-check* packet is *periodically* generated by the packet generator. If the sent replication packet has not yet been acknowledged by other switches, the ACK-check triggers its retransmission. Here we use the recirculation trigger for the packet generator to initiate a batch of packets at once.

In SRO, the packets themselves must be buffered since their content is not reproducible by the switch. Buffering in the data-plane is an open problem and we leave it for future work. However, since most NFs that use SRO would require the updates to be performed via the control-plane anyway (Observation 1, §4.1), we relay the reliable delivery to the control-plane of the switch that receives the write packet. The cost of buffering and retransmission is negligible, as we show in §9.2. Future switches might enable table updates in the
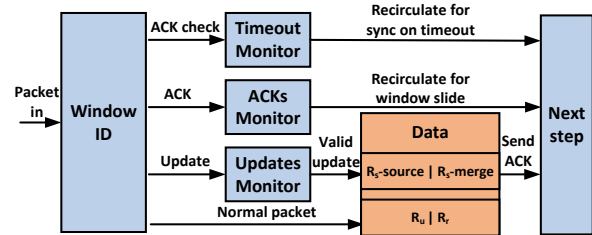
Figure 3: SDW high-level design. Blue boxes are reusable P4 control blocks, while the orange box is application-dependent.

data-plane, motivating data-plane buffering mechanisms to avoid control-plane involvement in replication.

Packet duplication and reordering. SRO replication packets are shipped with a sequence number allowing each replica to apply updates in order and to reject updates with sequence numbers lower than those already processed. In EWO, updates are idempotent and monotonic so detecting duplication and reordering is already a part of the merging process. We explain the SDW implementation in detail below.

7.2 Strong Delayed-Writes (SDW)

As presented in Figure 3, the data structure used in SDW is organized as two register arrays, each of which holds a 32-bit pair. At any given time, one window is designated for reading and writing, namely, its register arrays used as the R_u and R_r objects in the SDW protocol. The other window is used in the sync operation. The sync object R_s is divided into two register values, one, denoted R_s-merge, receives data from other switches, while the other, called R_s-source, holds the local state as sent to other switches at the beginning of the window. This separation is important to allow retransmissions (§7.1).

Synchronization. The alternating window structure enables SwiSh to ensure that the R_r in each window are consistent across all switches. Each time synchronization is initiated, the content of the R_s-source register is sent to all the switches, and the content received from all of the switches is merged in the R_s-merge register. Note that each switch also receives (and hence merges) its own update. Once all the updates are received, the content of R_s-merge is identical across the switches, so the synchronization for that window is finished.

Unfortunately, a full sketch cannot be read while processing a single packet, so we send the sketch column by column. For simplicity, we first explain handling of a single-column sketch, and then discuss the complete implementation.

Each switch maintains two bitmaps: one, to track ACKs that other switches received its updates, and the other, that it received all updates from them. If an update was lost, the sketch is retransmitted.

Window advancement. The last update to complete the bitmaps signifies the completion of the sync round for the

switch. The switch advances the window ID, swaps the roles of the registers, and starts a new sync process again. During this swap the following arrays are swapped: R_s-merge swaps with R_r, and R_s-source swaps with R_u.

Ready phase. Because round advancement is a local event, the switches do not advance their windows in lock-step. Thus, a switch may receive an update for the *next* window, which will be dropped and retransmitted later. Buffering such updates would significantly increase the memory footprint. Instead, we introduce the *ready phase*. Once a switch advances its local window it broadcasts a *ready* packet to all the rest. A switch starts broadcasting its updates only after it receives *ready* packets from all other switches (existing bitmap can be reused for tracking). This phase ensures that an update will not be sent to a switch that is not yet ready to merge it. Ready packets are retransmitted upon timeout, though in the experiments we did not encounter such cases. In our evaluation (§9.2) we show that the *ready* phase is critical to achieving predictable replication latency.

Multi-column sketches. Ideally, each switch should track each column being synchronized separately, to filter duplicates and retransmit lost updates. This solution would be too memory-consuming and would limit the sketch size, however. We make two optimizations. First, for R_s-merge, we retain the original bit-per-switch tracking, so a switch sends an ACK only when a full sketch was received. Thus, we always retransmit a full sketch. Second, we maintain a counter per switch which tracks the index of the next column to be updated. Only updates that match this counter are accepted. This approach is correct: it handles duplicates and packet reorders. However, while it is efficient for duplicates, it would lead to sketch retransmission if packets are reordered. We assume that this is a rare event, however, because IP routing in data centers usually maintains the same path for a given flow.

We implement both approaches. The bitmap-per-column implementation allows using sketches with 3 rows and 64K entries per-row and can scale up to 32 switches. The counter-per-switch implementation can scale to 4K switches for the same sketch size.

Note that changing the communication pattern from an all-to-all to an aggregation tree, e.g. as in SwitchML [68], may also reduce the per-switch state but at the cost of increasing replication latency.

Register initialization. There is no way to iterate over all the registers and reset them. Instead, we piggyback initialization on the first write and use a single bit in each register to determine whether the register is initialized. These bits are reset during the processing of sync packets.

Reducing replication bandwidth. Recall that SDW is used for a collection of variables, stored in register arrays, over which queries can take consistent snapshots. Our current implementation of the sync protocol exchanges a full state snapshot (including all variables) rather than only the ones that were updated. The challenge for selective updates is that the switches send a varying number of packets in each window (due to hardware limitations, the state does not fit in one packet), and so the destination does not know when to acknowledge the state receipt. To overcome this challenge, switches count the number of updates that they send in a window and piggyback this number on the last update.

Recovery. The recovery protocol follows the algorithm mentioned above (§6.4), but also considers the ready phase and sends ready packets to allow switches to make progress before removing the faulty switch from the replica group. SDW does not rely on a centralized controller in failure-free runs. However, as writes are not lost upon switch failures, recovery is kept off of the critical-path and is not time-sensitive. Therefore, we chose to offload the recovery protocol to a centralized controller which frees switch resources.

8 Implementation

We expound the implementation of SRO and EWO, and then we describe the distributed NFs implemented on top of SwiSh. Last, we provide additional implementation details and limitations.

8.1 Strong Read-Optimized (SRO)

We run the replication protocol in the control-plane logic. Write packets (packets that modify state) are forwarded to the control plane, which subsequently generates a write request forwards it to the head of the chain.

The way write requests are handled depends on the storage type where the data is stored in the switch. If the data can be modified only from the control-plane, then write requests must be processed by the control-plane at each switch in the chain. Otherwise, write requests can be processed directly in the data-plane. We implement reading from tail by tunneling the reading packets through the tail switch to its destination with an outer IP header (similar to IP-in-IP). While a write is pending, the key is flagged as "read-from-tail", causing subsequent reading packets to be sent to the tail.

8.2 Eventual Write-Optimized (EWO)

The EWO logic uses the following types of packets: (a) Regular packets from applications – read and write to the shared state. (b) Update packets – sent when the local state changes. The recipient merges these updates with its local state. (c) Generated packets – for reliable message delivery. Because each register array can only be accessed once per packet, if the state consists of an array, we generate one packet per array entry. If we maintain multiple register arrays, they can be accessed by a single packet.

Reads are local, while writes require sending an update to other switches. To broadcast updates, we use egress-to-egress mirroring to create a truncated copy of the original

write packet. We use the multicast engine to create a copy of the update packet for each switch in the replica group. Each copy is then modified to carry the updated values.

The application state each switch maintains depends on the particular data structure. For example, to implement a shared counter, each switch maintains a vector of counters, one per switch in the replica group. On the other hand, growing only sets and LWW variables do not require sharding.

In order to ensure eventual consistency in the face of lost update packets, a periodic background task is implemented by using the switch's packet generator that iterates over the register array, forming write update packets consisting of the indices and values for each register, and forwarding each one to a randomly-selected switch in the replica group.

8.3 Distributed NFs

We prototype three multi-switch NFs. We also prototype a distributed version for all of these NFs built using the protocols in SwiSh. In addition, for two of them we also implement a version that uses a central controller for synchronization.

Network Address Translator (NAT). This application maps internal source IPs to external source IPs. Each switch maintains two translation tables – one that maps (external source IP, external source port) to (internal source IP, internal source port) and another that performs the inverse mapping. We implement a distributed NAT using the SRO protocol. It requires no changes to the data-plane logic.

Super-spreaders detection (DDoS). This application detects source IPs that communicate with more than 1000 unique destination IPs. Inspired by OpenSketch [80], we implement it using a CMS, with a bit set instead of counters. Packets are first sampled based on the (source IP, destination IP) pair. Sampled packets set a single bit in the bitset in each row of the sketch. The bitset is used to estimate the number of unique destinations. Our implementation uses a sketch with 3 rows and 32K 32-bit wide bitsets per row.

We implement two designs based on a central controller. In both, the controller obtains the list of suspicious IPs from each switch, and decides to block IPs if the sum of different destination IPs for that source from both switches exceeds 1000, in which case it inserts an entry to the block list of each switch. However, there are two ways for the controller to obtain this data: (a) pull-based: each switch maintains a gradually growing list of potential IPs to block. The controller periodically *pulls* the delta in the list since the previous pull; (b) push-based: each switch sends a packet when it detects a potential IP to block. For simplicity we mark an IP as suspicious if it sends to more than 500 destinations, and construct the workload to send half of the packets from each source to one switch and another half to the other, thus the implementation works correctly for this case.

The distributed design replicates the sketch using the SDW protocol, each switch unilaterally decides to block a desti-

nation according to the replicated sketch, which essentially holds a global view of the network.

Rate limiter. We implement a rate limiter based on the token bucket algorithm [70]. In the single switch design, the controller periodically fetches rate estimations from each switch, calculates the token limit per each user and each switch, and writes it back to the switches. We implement two distributed versions, with EWO and SDW respectively. Switches replicate their own rate estimates for each user, and calculate their limit according to global traffic ratios.

8.4 Implementation Details

We implement SwiSh using $P4_{16}$ [73] and Intel P4 Studio 9.6.0 [32] for Tofino switches. We implement all protocols as described.

API. We expose the building blocks of each protocol's design as P4 control blocks [73]. We then use this API to implement our NF applications (§8.3).

Control Plane. For applications that use SRO variables, we implement the control-plane logic in C++ using the user space packet DMA API (kpkt). For the other protocols, we initialize the switch state using bfrt-python. We also utilize a simple TCP server in C++ for reading register values from the switch for the recovery protocol.

Limitations. Our current implementation does not include the required recovery logic for SRO because it is well-known and in-control plane, thus it does not challenge our design. Although independent to the number of switches in the replica group, the major limitation of replicated NFs is the increase in SRAM usage (×4). We fully implement recovery for SDW.

9 Evaluation

We evaluate the protocols and applications on two Tofino switches (each two pipes) and on 32 switches in an emulator. Our key observations are:

- Control-plane replication is too slow.
- SRO has high latency and low throughput.
- SDW is scalable and replicates large sketches in microseconds.
- For a DDoS detector, SDW responds instantly to an attack, blocking malicious packets, while central controller allows almost 50% of the packets to go through.
- For a rate limiter, SDW and EWO respond instantly to traffic changes, while central controller lags behind.

Setup. We use two machines with Intel Xeon Silver 4216 2.1 GHz CPUs, connected via two EdgeCore Wedge 100BF-32X programmable switches. The server is dual socket with 192 GB RAM. Hyper-threading and power saving are disabled. One machine acts as a traffic generator/consumer; it has two 100G Intel E800 NICs. The other acts as a central controller; it has two 40G NVIDIA ConnectX-4 Lx EN NICs.

Topology. We use the *leaf-spine* topology in which the switches are connected as shown in Figure 4, and run ECMP on one pipe and a NF on the second.
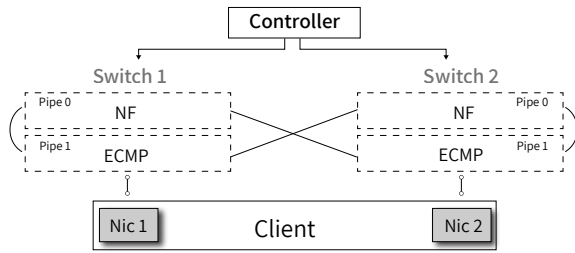
Figure 4: Testbed topology.

Performance measurement methodology. We build a DPDK-based packet generator. We evaluate SwiSh on a real packet trace, CAIDA [10], as well as synthetic workloads. Throughput is measured by the NIC and application-level performance counters. Latency is measured in software.

To measure the performance of the in-switch NFs and protocol implementations, we create a line-rate load (100Gbps, unless stated otherwise) on a single switch port. To validate that the performance obtained via this approach is representative of the switch under load over all its ports, we also run one experiment with *a fully loaded switch running at 2.1 Gpps* (§9.2). We show that the performance is almost the same as with a single port traffic, validating our methodology.

9.1 End-to-end benchmarks

NAT. We replay 10K packets from the CAIDA dataset and measure the per packet latency with and without replication. 21% of the packets are processed by the control plane (update packets), while the rest are processed in the data-plane. Figure 5d shows the latency distribution. SwiSh does not introduce any overheads for read packets, while update packets are taking about twice as long to get processed since they are batched in the control plane until the update is acknowledged by the other switch.

We also compare the throughput of the distributed version with the one on a single switch, while sending 64-byte packets at line rate to a single port. There are *no updates during the test*, as we wait for the handshake to complete. Therefore, both versions achieve line-rate throughput (112 Mpps).
Super-spreader detection. DDoS is configured to detect sources (IPs) that communicate with more than 1K different destinations. We create a trace where packets are sent from different source IPs, each with thousands of different destinations. Each source IP sends 10K packets.

In the experiment we replay a trace where we vary the number of packets that have different source IPs sent per second, while maintaining the absolute transfer rate from each source IP constant. This is a reasonable scenario where

an attacker uses a botnet to generate malicious traffic while maintaining the transfer rate of each bot constant.

We compare the number of packets sent by each source IP relative to the number of packets received by the destination IP. Ideally, each source IP should be blocked after the first 1000 packets, therefore the ratio should be about 10%.

We compare the push and pull baselines with the implementation that uses SDW replication. Figure 5b shows that both versions of the centralized controller are quickly becoming overwhelmed and cannot keep up with processing the updates, failing to block packets. At 1.5K source IPs/second the push baseline breaks down because the push requests to block certain IPs from the switch get dropped at the host, thus their respective IPs are left unblocked. The results were obtained after increasing the socket receive buffers to 25MB.

To validate this result, we run the same workload fixed at 4K source IPs/sec. Figure 5a shows the distribution of the ratio of packets received per source IP across all source IPs. We observe that the pull design manages to block up to 30% of all the source IPs, but for each IP different number of packets leaked. Effectively, the pull design was unable to block traffic from 70% of the source IPs. That is because the controller collects batches of requests and handles them together, thus some source IPs manage to send more than others. However, the push design blocks only 5% of all the source IPs. The SDW-based design, shown as a vertical line at 10%, passes the first 10% of each source (which is our super-spreader detection threshold), and then blocks all the packets as expected.
Per-user rate limiter. We set a limit of 2Mpps per-user and configure the rate limiter to re-estimate rates every 1ms.

We create a trace where packets are sent from different source IPs (each source defines a different user) with 40 unique users (sending rate is 2Mpps per user). The trace is comprised of alternating phases with a period of 5s. In even phases, all flows of a specific user are split equally between the two switches. While in odd phases, 90% of each user's flows are routed to one of the spine nodes and the rest 10% are forwarded to the other spine node. These alternations results in immediate changes in the per-user rate estimator that each switch maintains.

We compare our EWO and SDW protocols with a pull-based baseline and measure the average throughput per user over time. In the first 5 seconds of the experiment, the traffic is balanced so each switch runs at 1Mpps and the controller sets a per-user limit of 1Mpps on each switch. At the 5th second of the experiment, we change phases, and now one switch measures 1.8Mpps and the other switch measures 0.2Mpps. Because each switch was set to limit each user to 1Mpps, the first switch forwards only 1Mpps and the other switch forwards 0.2Mpps resulting in 1.2Mpps aggregate throughput. Figure 5c shows the average received throughput per-user over time at a sampling period of 200 ms. The baseline misses the phase changing point and allows the throughput to reduce to

(a) The distribution of the ratio of packets received per source IP across all 4K source IPs.

(b) The ratio of received packets for different number of DDoS sources per second.

(c) The impact of reaction time on a rate-limiter when changing traffic distribution.

(d) NAT latency distribution with and without replication.

(e) NAT latency with and without replication when sending update packets at a variable rate.

(f) The window advancement period for SDW replication for different state sizes.

Figure 5: End-to-end and analysis results.

below 1.5Mpps, slightly more than the expected value due to sampling. SDW and EWO perform compareably. They both react immediately to traffic changes without throughput drops. EWO eagerly sends 40 updates every 1ms, allowing the other switch to immediately change its limit. SDW replicates such a small state (40 32bit values) under 10 microseconds (5f).

9.2 Analysis

SRO: Update rate. We measure the overhead introduced to update packets with replication. For this experiment we run NAT on spine switches and send update packets that are all processed by the control plane. We measure the per packet latency and report the average. Figure 5e shows that compared to the single switch, in the replicated setting the update rate is reduced by a factor of ×2.5. This is expected since each update packet generates a write request and an ACK that has to be processed on the other switch.

We observe that *the update rate of SRO is limited by the throughput of the control plane on a single switch*. SRO variables cannot sustain more than 20K updates per second (160Kbps). Update latency increases with the update rate because packets are buffered in the control-plane until acknowledged.

SDW: Window advancement latency. We measure the time each window absorbs updates before being advanced. We vary the state size being replicated, so for the smallest sketch the time to advance the window is the upper bound on the replication rate, constrained by the latency of updates between switches. There are no retransmissions in this experiment.

We replicate a sketch with 3 rows and vary each row size to up to 64K counters (total of 768KB in each sketch). We store the global timestamps of the first 10K window increments and read them at the end of the run.

We use the following topologies: (a) *local-2*: two local pipes on a single switch (we measure identical results compared to two remote pipes); (b) *symmetric-4* - four pipes, two in each switch, with a dedicated link between each pipe.

Figure 5f shows the 99th percentile latency to advance the window for each state size. As we see, the window can be advanced as fast as every $3\mu sec$ for the scketch of 4 bytes. The current bottleneck is the packet sending rate which, even within the switch (Recirculation), takes a few hundred nanosecs. This window advancement rate implies that the updates become visible after $6\mu sec$ (since R_u becomes R_r after two window advancements). For the sketch larger than 1K, the actual replication rate is about *13Gbps* between each pair of switches, which is about *five orders of magnitude* faster than SRO. We note that this rate is limited by the maximum packet rate (∼160Mpps) of a single port. This is because replication packets hold only 12 bytes of data, which in turn is due to limited per-packet memory accesses imposed by the hardware. Optimizing the effective bandwidth is left for future work.

We observe negligible increase in the window advancement latency when adding two additional switches. This is because each switch updates all the others concurrently, hence no additional delay. The ready phase adds a constant latency overhead of $2\mu sec$ to each replication round.

SDW: Performance under full switch load. We generate

traffic on all switch ports as follows. We saturate a single port using our packet generation machine and let that traffic travel through each port in the switch by connecting ports in a chain and forming a "snake" (a similar methodology was used in NetCache [38]). We reserve ports that are used for replication. We use the symmetric-2 topology. We saturate the switch with 130B packets, each updating the sketch. For a sketch with 64K entries per-row we measure 486 μsec window advancement latency, at a total packet rate of 1.8Gpps. For a sketch size of 1 entry per-row we measure 3.2 μsec window advancement latency at a total packet rate of 2.1Gpps.[3] In both extremes, we could not measure any impact on window advancement latency, which is expected as the switch logic is guaranteed to perform packet processing at a switch line rate.

SDW: Retransmissions and the ready phase. The ready phase ensures that the switches do not send updates after advancing their window before all others advanced to that same window. Without this guarantee, an update from the consequent window that arrives too early will be dropped, and later retransmitted after a timeout. We now show that this phase is essential to avoid retransmissions and maintain low latency when scaling to more switches.

We first run the protocol without the ready phase (Figure 5f, local-2 no ready) on two pipes on the same switch. The protocol runs in lockstep on both of the pipes, so we do not see any update retransmissions. However, with four pipes (symmetric-4 topology) there are many retransmission (not shown in the Figure). For example, we measure an average of 2934 update retranmissions in the first 10K window advancements across 100 runs. We observe a similar behavior in an asymmetric topology four pipes connected using the leaf-spine topology. Adding the ready phase completely eliminates such retransmissions and allows the system to progress effectively as fast as a two-pipes system, with stable latency guarantees.

SDW: Recovery. We measure the total recovery time of the protocol from the time pipes fail to the time the system makes progress, i.e. windows are advancing again. We run four pipes in the 4-symmetric topology that replicate a sketch and shut down random pipes. We disable the failed pipes' ports to other switches in a random order. We repeat this experiment 20 times for each data point, and vary sketch sizes and failure counts. We report the average recovery time.

Figure 6 shows that recovery time is dominated by the time it takes to synchronize the sketches of correct switches. Therefore, recovery time increases as sketch size increases, and decreases as the failure count increases. As expected, for the 3 pipe failures setup, only a single correct switch remains live, thus recovery time is independent from the sketch size.

As explained in §6.4, updates sent to live switches during the recovery are not lost but accumulated, so the recovery time minimization is a secondary goal. Nevertheless, recovery time can be further reduced by applying additional optimizations,

---

[3]1-entry per-row requires lower replication load and frees certain resources affording higher packet rate.
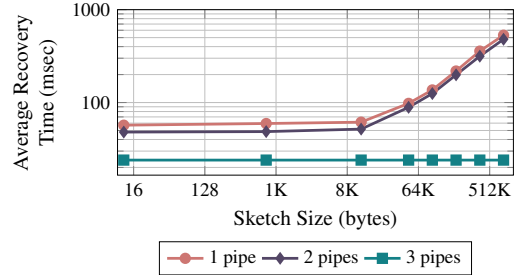
Figure 6: The impact of number of failures and sketch size on the total SDW recovery time.

e.g. parallelizing the currently serial controller-to-switch communication and batching requests, and by writing the logic using a more efficient programming language.

SDW: Scalability. We emulate a large replica group of switches running the SDW protocol by connecting together 32 Tofino model instances running in Docker containers. The switches are connected together via another switch that runs L3 forwarding. We verify that the protocol runs correctly and that there are no update retransmissions.

10 Related Work

In-switch NFs. Previous studies have shown that offloading NFs to programmable switches, such as load balancers [57] and DDoS detectors [45], enables very high performance. However, these projects were designed for a single switch. SwiSh aims to facilitate the deployment of these applications in a distributed fashion. RedPlane [42] enables switch state replication to servers for fault tolerance, but does not support state modification on multiple switches concurrently, as our work does.

In-switch acceleration. Previous works suggested in-switch acceleration for general-purpose applications such as key-value caches [38, 50], replicated key-value stores [37], query processing [24] and aggregations [68, 75]. SwiSh can be useful for such general-purpose applications too. For example, SwiSh could be used to implement the cache invalidation mechanism in DistCache. We note, however, that due to the general-purpose nature of these applications, some of them feature a complex state, and require strong semantics together with frequent updates, which SwiSh does not provide. Such requirements are less common in NFs; thus, we target SwiSh to facilitate the development of distributed NFs.

State management for NFs. State management and fault-tolerance for NFs on servers have been well studied [20, 64, 65, 71, 77]. However, these techniques are infeasible in the context of programmable switches. For example, FTMB [71] suggests a rollback-recovery technique for fault-tolerance in which packets are logged and replayed upon failures. However, due to the high processing rate of the switch, it is impractical to log

every packet to external storage or through the control-plane.

In-switch coordination. NetChain [37] and P4xos [16] implement coordination protocols running in the data plane to provide reliable storage as a network service. We apply data plane replication as an internal building block for NFs, a task for which it is well suited as the data-plane properties (e.g., limitations to ~100 byte objects) are better matched for replicating NF state registers than arbitrary applications.

Distributed network state. Managing distributed network state has been well studied. Onix [43] distributes network-wide state among multiple controllers. DIFANE [81] offloads forwarding decisions to authority switches to alleviate load on the controller and to reduce per-flow memory usage in network switches. Mahajan *et al.* [55] explore consistency semantics during network state updates. While previous works focus on control-plane managed state, SwiSh specifically targets replication of mutable state of data-plane programs.

Distributed network monitoring. Network-wide monitoring requires coordinated, distributed computation across switches [25, 26, 63]. Harrison *et al.* [25, 26] propose a distributed heavy-hitter detection algorithm that combines local counters with a centralized controller. SwiSh can be used to implement similar algorithms without a centralized controller, potentially providing faster response. Ripple [36] replicates state in data-plane for link-flooding defense but does not provide consistency guarantees. Ripple can be implemented using SwiSh. Distributed computation is also needed if the resources of a single switch are insufficient, e.g. Demianiuk *et al.* [18] partition state across switches for flow metric computation.

Relaxing consistency for availability. Many systems have traded consistency for increased availability and performance [4, 17, 21, 44, 62, 72, 79]. For example, TACT [79] aims to provide a middle-ground between strong and eventual consistency. However, TACT may block read and write operations to enforce consistency bounds which is unsustainable in the switch environment. Additionally, TACT maintains a single version of the data while SDW maintains multiple versions of the state and seamlessly switches to the up-to-date one as soon as the previous synchronization round is completed. Therefore, the protocol advances as fast as the network conditions allow while providing consistent snapshots to every replica. On the other hand, the combination of dynamic system behavior and consistent snapshots cannot be expressed using TACT's consistency metrics.

11 Conclusions

SwiSh offers a systematic approach to state sharing among programmable switches. We analyze the requirements of in-switch stateful NFs and implement three protocols for data-plane replication. We introduce a novel SDW protocol that achieves high update rate and low update latency, while providing strong consistency guarantees, which are particularly useful for implementing sketches. We show experimentally that data-plane is practical and fast, and achieves orders of magnitude higher performance than the traditional centralized controller designs. We believe that this work will pave the way for building distributed stateful NFs entirely in data-plane.

Acknowledgments

References

[1] Yehuda Afek, Anat Bremler-Barr, Shir Landau Feibish, and Liron Schiff. Detecting heavy flows in the SDN match and action model. *Comput. Networks*, 136:1–12, 2018.

[2] Pankaj K Agarwal, Graham Cormode, Zengfeng Huang, Jeff M Phillips, Zhewei Wei, and Ke Yi. Mergeable summaries. *ACM Transactions on Database Systems (TODS)*, 38(4):1–28, 2013.

[3] Amazon Web Services. AWS Shield. https://aws.amazon.com/shield.

[4] Mary Baker and John Ousterhout. Availability in the Sprite distributed file system. In *Proceedings of the 4th workshop on ACM SIGOPS European workshop*, pages 1–4, 1990.

[5] Ziv Bar-Yossef, TS Jayram, Ravi Kumar, D Sivakumar, and Luca Trevisan. Counting distinct elements in a data stream. In *International Workshop on Randomization and Approximation Techniques in Computer Science*, pages 1–10. Springer, 2002.

[6] Ran Ben Basat, Xiaoqi Chen, Gil Einziger, Shir Landau Feibish, Danny Raz, and Minlan Yu. Routing oblivious measurement analytics. In *IFIP Networking*, pages 449–457, 2020.

[7] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, et al. ONOS: Towards an Open, Distributed SDN OS. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 1–6, 2014.

[8] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica,

and Mark Horowitz. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. *ACM SIGCOMM Computer Communication Review*, 43(4):99–110, 2013.

[9] Broadcom. Trident 3. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56870-series/.

[10] CAIDA. The CAIDA UCSD Anonymized Internet Traces - 2019. https://www.caida.org/catalog/datasets/passive_dataset.

[11] Martin Casado, Michael J Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking Control of the Enterprise. *ACM SIGCOMM computer communication review*, 37(4):1–12, 2007.

[12] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, and Ori Rottenstreich. Catching the Microburst Culprits with Snappy. In *Proceedings of the Afternoon Workshop on Self-Driving Networks*, SelfDN 2018, page 22–28, New York, NY, USA, 2018. Association for Computing Machinery.

[13] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, Ori Rottenstreich, Steven A. Monetti, and Tzuu-Yi Wang. Fine-grained queue measurement in the data plane. In *ACM SIGCOMM Conference on Emerging Networking EXperiments and Technologies*, pages 15–29. ACM, 2019.

[14] Graham Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, April 2005.

[15] Graham Cormode and S. Muthu Muthukrishnan. Approximating data with the count-min sketch. *IEEE Software*, 29(1):64–69, 2012.

[16] H. T. Dang, P. Bressana, H. Wang, K. S. Lee, N. Zilberman, H. Weatherspoon, M. Canini, F. Pedone, and R. Soulé. P4xos: Consensus as a Network Service. *IEEE/ACM Transactions on Networking*, pages 1–13, 2020.

[17] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.

[18] V. Demianiuk, S. Gorinsky, S. Nikolenko, and K. Kogan. Robust Distributed Monitoring of Traffic Flows. In *2019 IEEE 27th International Conference on Network Protocols (ICNP)*, pages 1–11, 2019.

[19] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A Fast and Reliable Software Network Load Balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 523–535, Santa Clara, CA, 2016.

[20] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. OpenNF: Enabling Innovation in Network Function Control. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 163–174, New York, NY, USA, 2014. Association for Computing Machinery.

[21] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, 2003.

[22] Wojciech Golab, Lisa Higham, and Philipp Woelfel. Linearizable implementations do not suffice for randomized distributed computation. In *Proceedings of the forty-third annual ACM symposium on Theory of computing*, pages 373–382, 2011.

[23] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: A scalable and flexible data center network. In *Proceedings of ACM SIGCOMM 2009*, Barcelona, Spain, August 2009. ACM.

[24] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-Driven Streaming Network Telemetry. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 357–371, New York, NY, USA, 2018. Association for Computing Machinery.

[25] Rob Harrison, Shir Landau Feibish, Arpit Gupta, Ross Teixeira, S. Muthukrishnan, and Jennifer Rexford. Carpe elephants: Seize the global heavy hitters. In *Proceedings of the 2020 ACM SIGCOMM 2020 Workshop on Secure Programmable Network Infrastructure, SPIN@SIGCOMM 2020, Virtual Event, USA, August 14, 2020*, pages 15–21, 2020.

[26] Harrison, Rob and Cai, Qizhe and Gupta, Arpit and Rexford, Jennifer. Network-Wide Heavy Hitter Detection with Commodity Switches. In *Proceedings of the Symposium on SDN Research*, SOSR '18, New York, NY, USA, 2018. Association for Computing Machinery.

[27] Thomas A Henzinger, Christoph M Kirsch, Hannes Payer, Ali Sezgin, and Ana Sokolova. Quantitative relaxation of concurrent data structures. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 317–328, 2013.

[28] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[29] Qun Huang, Xin Jin, Patrick P. C. Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. SketchVisor: Robust network measurement for software packet processing. In *ACM SIGCOMM*, pages 113–126, 2017.

[30] Qun Huang, Patrick P. C. Lee, and Yungang Bao. Sketchlearn: Relieving user burdens in approximate measurement with automated statistical inference. In *ACM SIGCOMM*, pages 576–590, 2018.

[31] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8, 2010.

[32] Intel. P4 Studio. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/p4-suite/p4-studio.html.

[33] Intel. Tofino. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html.

[34] Intel. Tofino Native Architecture. https://github.com/barefootnetworks/Open-Tofino.

[35] Nikita Ivkin, Zhuolong Yu, Vladimir Braverman, and Xin Jin. Qpipe: Quantiles sketch fully in the data plane. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, pages 285–291, 2019.

[36] Jiarong Xing and Wenqing Wu and Ang Chen. Ripple: A Programmable, Decentralized Link-Flooding Defense Against Adaptive Adversaries. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3865–3881. USENIX Association, August 2021.

[37] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. NetChain: Scale-Free Sub-RTT Coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 35–49, Renton, WA, April 2018. USENIX Association.

[38] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 121–136, New York, NY, USA, 2017. Association for Computing Machinery.

[39] Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. Stateless Network Functions: Breaking the Tight Coupling of State and Processing. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 97–112, Boston, MA, March 2017. USENIX Association.

[40] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. HULA: Scalable Load Balancing Using Programmable Data Planes. In *Proceedings of the 2016 Symposium on SDN Research (SOSR '16)*, Santa Clara, CA, USA, March 2016. ACM.

[41] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the 1994 USENIX Winter Technical Conference*, San Francisco, CA, USA, January 1994. USENIX.

[42] Daehyeok Kim, Jacob Nelson, Dan R. K. Ports, Vyas Sekar, and Srinivasan Seshan. RedPlane: Enabling Fault-Tolerant Stateful in-Switch Applications. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 223–244, New York, NY, USA, 2021. Association for Computing Machinery.

[43] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, et al. Onix: A Distributed Control Platform for Large-Scale Production Networks. In *OSDI*, volume 10, pages 1–6, 2010.

[44] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[45] A. C. Lapolli, J. Adilson Marques, and L. P. Gaspary. Offloading Real-time DDoS Attack Detection to Programmable Data Planes. In *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 19–27, 2019.

[46] Kfir Lev-Ari, Edward Bortnikov, Idit Keidar, and Alexander Shraer. Composing ordered sequential consistency. *Information Processing Letters*, 123:47–50, 2017.

[47] B. Lewis, M. Broadbent, and N. Race. P4ID: P4 Enhanced Intrusion Detection. In *2019 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 1–4, 2019.

[48] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[49] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. FlowRadar: A better NetFlow for data centers. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 311–324, Santa Clara, CA, March 2016.

[50] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. DistCache: Provable Load Balancing for Large-Scale Storage Systems with Distributed Caching. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies*, FAST'19, page 143–157, USA, 2019. USENIX Association.

[51] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. Nitrosketch: Robust and general sketch-based monitoring in software switches. In *ACM SIGCOMM*, pages 334–350, 2019.

[52] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with UnivMon. In *ACM SIGCOMM*, pages 101–114, 2016.

[53] Zaoxing Liu, Hun Namkung, Georgios Nikolaidis, Jeongkeun Lee, Changhoon Kim, Xin Jin, Vladimir Braverman, MinlanYu, and Vyas Sekar. Jaqen: A high-performance switch-native approach for detecting and mitigating volumetric ddos attacks with programmable switche. In *Proc. USENIX Security*, 2021.

[54] Zaoxing Liu, Samson Zhou, Ori Rottenstreich, Vladimir Braverman, and Jennifer Rexford. Memory-efficient performance monitoring on programmable switches with lean algorithms. In *Symposium on Algorithmic Principles of Computer Systems, APOCS*, pages 31–44, 2020.

[55] Mahajan, Ratul and Wattenhofer, Roger. On Consistent Updates in Software Defined Networks. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, HotNets-XII, New York, NY, USA, 2013. Association for Computing Machinery.

[56] McKeown, Nick and Anderson, Tom and Balakrishnan, Hari and Parulkar, Guru and Peterson, Larry and Rexford, Jennifer and Shenker, Scott and Turner, Jonathan. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[57] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 15–28, 2017.

[58] Microsoft Azure. Azure DDoS Protection. https://azure.microsoft.com/en-us/services/ddos-protection/.

[59] Robert HB Netzer and Jian Xu. Necessary and sufficient conditions for consistent global snapshots. *IEEE Transactions on Parallel and distributed Systems*, 6(2):165–169, 1995.

[60] NVIDIA. Spectrum. https://www.nvidia.com/en-us/networking/ethernet-switching/spectrum-sn4000/.

[61] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. Ananta: Cloud Scale Load Balancing. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 207–218, New York, NY, USA, 2013. Association for Computing Machinery.

[62] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible Update Propagation for Weakly Consistent Replication. *SIGOPS Oper. Syst. Rev.*, 31(5):288–301, oct 1997.

[63] Raghavan, Barath and Vishwanath, Kashi and Ramabhadran, Sriram and Yocum, Kenneth and Snoeren, Alex C. Cloud Control with Distributed Rate Limiting. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '07, page 337–348, New York, NY, USA, 2007. Association for Computing Machinery.

[64] Shriram Rajagopalan, Dan Williams, and Hani Jamjoom. Pico Replication: A High Availability Framework for Middleboxes. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[65] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 227–240, Lombard, IL, 2013. USENIX.

[66] Arik Rinberg and Idit Keidar. Intermediate value linearizability: A quantitative correctness criterion. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*, volume 179 of *LIPIcs*, pages 2:1–2:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[67] Arik Rinberg, Alexander Spiegelman, Edward Bortnikov, Eshcar Hillel, Idit Keidar, Lee Rhodes, and Hadar Serviansky. Fast concurrent data sketches. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '20, pages 117–129, 2020.

[68] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling distributed machine learning with In-Network aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 785–808. USENIX Association, April 2021.

[69] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-Free Replicated Data Types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, page 386–400, Berlin, Heidelberg, 2011. Springer-Verlag.

[70] S. Shenker and J. Wroclawski. RFC2215: General Characterization Parameters for Integrated Service Network Elements, 1997.

[71] Justine Sherry, Peter Xiang Gao, Soumya Basu, Aurojit Panda, Arvind Krishnamurthy, Christian Maciocco, Maziar Manesh, João Martins, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. Rollback-Recovery for Middleboxes. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 227–240, New York, NY, USA, 2015. Association for Computing Machinery.

[72] Jeff Terrace and Michael J. Freedman. Object Storage on CRAQ: High-Throughput Chain Replication for Read-Mostly Workloads. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, page 11, USA, 2009. USENIX Association.

[73] The P4 Language Consortium. P4$_{16}$ Language Specification. `https://p4.org/p4-spec/docs/P4-16-v1.2.0.html`.

[74] The P4.org Architecture Working Group. P4$_{16}$ Portable Switch Architecture (PSA). `https://p4.org/p4-spec/docs/PSA-v1.1.0.html`.

[75] Muhammad Tirmazi, Ran Ben Basat, Jiaqi Gao, and Minlan Yu. Cheetah: Accelerating Database Queries with Switch Pruning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 2407–2422, New York, NY, USA, 2020. Association for Computing Machinery.

[76] Robbert van Renesse and Fred B. Schneider. Chain Replication for Supporting High Throughput and Availability. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation*, OSDI'04, page 7, USA, 2004. USENIX Association.

[77] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. Elastic Scaling of Stateful Network Functions. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 299–312, Renton, WA, April 2018. USENIX Association.

[78] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 561–575, 2018.

[79] Haifeng Yu and Amin Vahdat. Design and Evaluation of a Continuous Consistency Model for Replicated Services. OSDI'00, USA, 2000. USENIX Association.

[80] Minlan Yu, Lavanya Jose, and Rui Miao. Software defined traffic measurement with opensketch. In Nick Feamster and Jeffrey C. Mogul, editors, *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*, pages 29–42, April 2013.

[81] Minlan Yu, Jennifer Rexford, Michael J Freedman, and Jia Wang. Scalable flow-based networking with DIFANE. *ACM SIGCOMM Computer Communication Review*, 40(4):351–362, 2010.

[82] Zeno, Lior and Ports, Dan R. K. and Nelson, Jacob and Silberstein, Mark. SwiShmem: Distributed Shared State Abstractions for Programmable Switches. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, HotNets '20, page 160–167, New York, NY, USA, 2020. Association for Computing Machinery.

[83] Menghao Zhang, Guanyu Li, Shicheng Wang, Chang Liu, Ang Chen, Hongxin Hu, Guofei Gu, Qianqian Li, Mingwei Xu, and Jianping Wu. Poseidon: Mitigating volumetric ddos attacks with programmable switches. In *the 27th Network and Distributed System Security Symposium (NDSS 2020)*, 2020.

A Theoretical Analysis

The SDW protocol supports stream-order invariant data types like sketches. Variables of this type support three API functions: (1) UPDATE(v) – handling a single addition of element v, (2) QUERY() – returns a value based on the internal state, and MERGE(R') – merges the state of R' with that of the current variable. A requirement of any variable R fitting this model is that the QUERY result depends only on the set of elements that were ingested before it (either by an UPDATE or a MERGE), and not their order. We say that a query *reflects* an update, if the update altered the state before the query executed.

An execution of an algorithm renders a *history H*, which is a series of *invoke* and *response* events of the three API functions. In a *sequential history* each invocation is immediately followed by its response. The *sequential specification $\mathcal{H}$* of a variable is its set of allowed sequential histories.

A *linearization* of a concurrent execution σ is a history $H \in \mathcal{H}$ such that after adding responses to some pending invocations and removing others, H and σ consist of the same invocations and responses and H preserves the order between non-overlapping operations [28]. If every concurrent execution has a linearization, we say that the variable is linearizable. For randomized variables we require a stronger property, called *strong linearizability*. The qualifier "strong" means that the linearization points are not determined post-facto, which is necessary in randomized variables [22].

A relaxed property of a variable is an extension of its sequential specification to allow for more behaviors. We adopt the notion of *r-relaxed strong linearizability* from [67], a variant of the relaxation defined by Henzinger et al. [27], brought here for completeness. Intuitively, an r-relaxed variable allows a query to return a result based on all but at most r updates that happened before it.

Definition A.1. A sequential history H is an r-relaxation of a sequential history H', if H is comprised of all but at most r of the invocations in H' and their responses, and each invocation in H is preceded by all but at most r of the invocation that precede the same invocation in H'. The r-relaxation of $\mathcal{H}$ is the set of histories that have r-relaxations in $\mathcal{H}$, denoted $\mathcal{H}^r$.

Our SDW protocol is described in §6.3, and its pseudo-code is presented in Algorithm 1. To prove that Algorithm 1 is r-relaxed strongly linearizable, we first prove a helper lemma:

Lemma 1. *Consider a history H arising from a concurrent execution of Algorithm 1, and some completed update $u \in H$ executed by p_i. Let w be the value of win during u. Update u is reflected by every query q on any p_j, in every window $w' \geq w + 2$.*

Proof. Let H be a history arising from a concurrent execution of Algorithm 1, and let $u \in H$ be some completed update executed by p_i. Let w be the value of *win* during the update's execution on p_i.

Update u is added to $objs[w \mod 3]$ on Line 12. On Line 39, $objs[(w+2) \mod 3]$ is broadcast to all switches, specifically to some switch p_j (as p_i retains the update in the same place that is merges received variables, this holds for $j = i$).

The next time p_j advances on Line 35, it enters window $w' = w + 2$. Note that the variable that was queried in the previous window ($w' - 1$) is the same variable that reflected u. This variable is the one queried in round w', therefore reflected in round $w' = w + 2$.

We now prove by induction that in round $w'' = w' + k$, u is reflected by a query in round w'' on p_j. The base is for $k = 0$, and has been prove.

Assume the hypothesis holds for $w' + l$, we prove for $w' + l + 1$. In round $w' + l$, u is reflected by $obj[(w' + l + 1) \mod 3]$. On Line 37, p_j merges this variable into $obj[((w' + l + 1) + 1) \mod 3]$, which is the variable queried in this round.

As this induction is true for all $k \geq 0$, it holds for any $w'' \geq w'$, proving the lemma. □

The following corollary follows directly from Lemma 1:

Corollary 1.1. *Let H be a history arising from a concurrent execution of Algorithm 1, and let $q \in H$ be some query completed by p_i. Let w be the value of win during its execution. Query q reflects all updates occurring in any window $w' \leq w - 2$.*

Note: A system where linearizability holds for sub histories including a single query is sometimes called *Ordered Sequential Consistency (OSC)* [46], this is commonly used in systems, e.g., ZooKeeper [31].

Finally, we define the *operation projection* of a history H and a set of operations O as the same history containing only invocations and responses of operations in O. We denote this $H|_O$ Using these formalisms we can prove the following theorem:

Theorem 2. *Consider a history H arising from a concurrent execution of Algorithm 1, and some query $q \in H$. Let U be the set of updates in H. The history of $H|_{U \cup \{q\}}$ is r-relaxed strongly linearizable.*

Proof. Let H be a history arising from a concurrent execution of Algorithm 1, let $q \in H$ be some query by p_i, and let U be the set of all updates in H. Denote $H|_{U \cup \{q\}}$ as H'. We show that H' is r-relaxed strongly linearizable with respect to $\mathcal{H}^r$, for $r = 2NB$. To prove this, we show the existing of two mappings, f and g, such that f maps operations in H' to *visibility points*, and g maps operations in H' to linearization points. Intuitively, visibility points are the time in the execution when an update is visible to a query, i.e., the query reflects the update. Bounding the number of preceding but not yet visible updates gives the relaxation.

We show that (1) $f(H') \in \mathcal{H}$, and (2) $g(H')$ is an r relaxation of $f(H')$. Together, this implies the theorem.

The visibility points ($f(H')$) are as follows:

- For the query, its visibility point is its return.
- For an update returning *false* at time t, its visibility point is t.
- For an update returning *true* at time t, let w be p_i's value of *win* at time t. The visibility point is the first time after t that p_i's value of *win* is $win + 2$.

Note that in the latter case, the visibility point is after the update returns, so f does not preserve real-time order.

The linearization points ($g(H')$) are as follows:

- An update's linearization point is its return, either *true* or *false*.
- A query's linearization point is its return.

By definition, the linearization points as defined by $g(H')$ aren't decided post-facto – rather the linearization is a pre-determined point in the execution.

Consider some update $u \in H'$ executed on some p_j that returns *true*. Let w be p_j's value of *win* during its execution. Let w' be p_i's value of *win* during q's execution. We show that if $w \leq w' - 2$, then q observes u, and if $w > w' - 2$, then q doesn't observe u.

From the definition of Algorithm 1, for any win_i on p_i and win_j on p_j, $|win_i - win_j| \leq 1$.

If $w = w' - 2$, then when p_j added u to its local buffers, it did so to $obj[w \mod 3]$. As $|win_i - win_j| \leq 1$, p_j advanced at least 1 window from w. When it did so, it sent $obj[w \mod 3]$ to p_i. In window $w' - 1$, p_i merges the update into $obj[w' + 1 \mod 3]$. In window w' this same variable is queried, thus q observes u. If $w \leq w' - 3$, then the update is merged into some index of the variables array, and is copied over until it is reflected in all 3 of them, and specifically reflected in $obj[w' + 1 \mod 3]$ in window w'.

If $w \geq w' - 1$, then when p_j added u into its local buffer it did so to $obj[w \mod 3]$. This update is sent to p_i only in window $w + 1$, and therefore isn't reflected in $obj[w' + 1 \mod 3]$ in window w'.

Therefore, q reflects all updates that return true that happened during any window $w \leq w' - 2$. As there are at most B updates that return true in any window, q reflects all but at most $2NB$ updates that precede it in H. Therefore, $g(H')$ is an $2NB$-relaxation of $f(H')$.

As the query returns a value based on the updates that happened before it, and each access to the process local state is down sequentially, q returns a value that reflects all successful updates that happen before it in $f(H')$. Therefore, $f(H') \in \mathcal{H}$. □

Intuitively, every query returns a value reflecting a sub-stream of its preceding and concurrent updates, consisting of all but at most r successful ones. The upper bound r on the number of "missing" updates is of vast importance, without it

the drift between one switch and another can grow in an unbounded fashion. For example, consider a counter distributed among two switches running an eventually synchronous algorithm. One switch can increment the counter an arbitrarily large number of times, while the other returns 0 on every query – the promise of eventual synchrony is too weak.

Theorem 2 ensures that every history consisting of a single query and all updates is r-relaxed strongly linearizable, which in many cases preserves some relaxation of the error bounds. For example, Rinberg et al. [67] show that, under a weak adversary, a K-Minimum Value (KMV) θ sketch [5] has an error of at most twice that of the sequential one. Another example is a relaxed Quantiles sketch [2], which has an additive error of $r/n - (r\epsilon)/n$ with some tuning parameter ϵ, where r is the relaxation and n is the stream size. Thus, the impact of the relaxation diminishes as the stream size grows.

Algorithm 1: Algorithm running on switch p_i.

```
1  initialization:
2  win ← 0
3  count ← 0
4  objs ← [obj.init(), obj.init(), o.init()]
5  buf ← {}
6  rcvs ← {}
7  acks ← {}
8  Function Update(v):
9      if count == B then
                                                              // Write variable is full
10         return false
11     else
12         objs [win mod 3].update(v)                        // Add to the write variable
13         count ← count + 1
14         return true

15
16 Function Query():
17     return objs [(win + 1) mod 3].query()                 // Serve query from read variable

18
19 on receive "(o', w')" from pj:
                                                              // Sync
20     if w' > win then
21         buf ← buf ∪ {(o', w')}                            // Buffer messages from future windows
22     else
23         rcvs ← rcvs ∪ {j}
24         objs [(win + 2) mod 3].merge(o')                  // Merge into sync buffer
25         send "ack" to pj
26         check_done()

27
28 on receive "ack" from pj:
29     acks ← acks ∪ {j}
30     check_done()

31
32 Function check_done():
33     if |rcvs| == n && |acks| == n then
34         count ← 0
35         win ← win + 1                                     // Rotate right
36         o' ← objs [win mod 3]
37         objs [(win + 1) mod 3].merge(o')                  // Add the updates from window w to the current state
38         objs [win mod 3] ← o.init()                       // Clear write variable
39         broadcast "(objs [(win + 2) mod 3], win)"         // Send sync message
40         rcvs ← {i}
41         acks ← {i}
42         forall (o', w') in buf do
                                                              // Handle buffered messages
43             rcvs ← rcvs ∪ {j}
44             objs [(win + 2) mod 3].merge(o')
45             send "ack" to pj
46         buf ← {}
```

Modular Switch Programming Under Resource Constraints

Mary Hogan[1], Shir Landau-Feibish[2], Mina Tahmasbi Arashloo[3], Jennifer Rexford[1], and David Walker[1]

[1]Princeton University
[2]The Open University of Israel
[3]Cornell University

Abstract

Programmable networks support a wide variety of applications, including access control, routing, monitoring, caching, and synchronization. As demand for applications grows, so does resource contention within the switch data plane. Cramming applications onto a switch is a challenging task that often results in non-modular programming, frustrating "trial and error" compile-debug cycles, and suboptimal use of resources. In this paper, we present P4All, an extension of P4 that allows programmers to define *elastic* data structures that stretch automatically to make optimal use of available switch resources. These data structures are defined using *symbolic primitives* (that parameterize the size and shape of the structure) and *objective functions* (that quantify the value gained or lost as that shape changes). A top-level optimization function specifies how to share resources amongst data structures or applications. We demonstrate the inherent modularity and effectiveness of our design by building a range of reusable elastic data structures including hash tables, Bloom filters, sketches, and key-value stores, and using those structures within larger applications. We show how to implement the P4All compiler using a combination of dependency analysis, loop unrolling, linear and non-linear constraint generation, and constraint solving. We evaluate the compiler's performance, showing that a range of elastic programs can be compiled to P4 in few minutes at most, but usually less.

1 Introduction

P4 has quickly become a key language for programming network data planes. Using P4, operators can define their own packet headers and specify how the data plane should parse and process them [7]. In addition to implementing traditional forwarding, routing, and load-balancing tasks, this flexibility has enabled new kinds of in-network computing that can accelerate distributed applications [26, 27] and perform advanced monitoring and telemetry [10, 11, 17, 30].

All of these applications place demands on switch resources, but for many, the demands are somewhat flexible:

additional resources, typically memory or stages in the PISA pipeline, improve application performance, but do not necessarily make or break it. For instance, NetCache [27] improves throughput and latency for key-value stores via in-network computing. Internally, it uses two main data structures: a count-min sketch (CMS) to keep track of popular keys, and a compact key-value store (KVS) to maintain their corresponding values. Increasing or decreasing the size of those structures will have an impact on performance, but does not affect the correctness of the system—a cache miss may increase latency, but the correct values will always be returned for a given key. Other applications, such as traffic-monitoring infrastructure, have similar properties. Increasing the size of the underlying hash tables, Bloom filters, sketches, or key-value stores may make network monitoring somewhat more precise but does not typically result in all-or-nothing decisions.

Because resource constraints for these components are flexible, network engineers can, in theory, squeeze multiple different applications onto a single device. Unfortunately, however, doing so using today's programming language technology is a challenging and error-prone task: P4 forces programmers to hardcode their decisions about the size and shape of their data structures. If the data structure is too large, the program simply fails to compile and little feedback is provided; if it is too small, it will compile but the resources will be used suboptimally. Moreover, structures are not reuseable: a cache, that fits just fine on a switch alongside a table for IP forwarding, is suddenly too large when a firewall is added. To squeeze the cache in, programmers may have to rewrite the internals of their cache, manually adjusting the number or sizes of the registers or match-action tables used. To test their work, they resort to a tedious trial-and-error cycle of rewriting their applications, and invoking the compiler to see if it can succeed in fitting the structures into the available hardware resources.

This manual process of tweaking the *internal* details of data structures, and checking whether the resulting structures satisfy *global* constraints, is inherently non-modular: Programmers tasked with implementing separate applications cannot do so independently. Indeed, while the same data structures

Data Structure	Used in
Key-value store/ hash table	Precision [6], Sonata [17], Network-Wide HH [19], Carpe [20], Sketchvisor [23], LinearRoad [25], NetChain [26], NetCache [27], FlowRadar [30], HashPipe [41], Elastic Sketch [46]
Hash-based matrix (Sketch)	AROMA [4], Sketchvisor [23], Sketchlearn [24], NetCache [27], Nitrosketch [31], UnivMon [32], Sharma et al. [38], Fair Queueing [39], Elastic Sketch [46]
Bloom filter	NetCache [27], FlowRadar [30], SilkRoad [34], Sharma et al. [38]
Multi-value table	BeauCoup [10], Blink [22]
Sliding window sketch	PINT [5], Conquest [11]
Ring buffer	NetLock [47], Netseer [48]

Figure 1: PISA data structures

appear again and again (see Figure 1 for a selection), the varying resource constraints makes it difficult to reuse these structures for different targets or applications.

Elastic Switch Programming. We extend P4 with the ability to write *elastic* programs. An elastic program is a single, compact program that can "stretch" to make use of available hardware resources or "contract" to squeeze in beside other applications. Elastic programs can be constructed from any number of elastic components that each stretch arbitrarily to fill available space. An elastic NetCache program, for example, may be constructed from an elastic count-min sketch and an elastic key-value store. The programmer can control the relative stretch of these modules by specifying an objective function that the compiler should maximize. For example, the NetCache application could maximize the cache "hit rate" by prioritizing memory allocation for the key-value store (to store more of the "hot" keys) while ensuring that enough remains for the count-min sketch to produce sufficiently accurate estimates of key popularity. In addition to memory, programs could simultaneously maximize the use of other switch resources such as available processing units and pipeline stages.

To implement these elastic programs, we present P4All, a backward-compatible extension of the P4 language with several additional features: (1) symbolic values, (2) symbolic arrays, (3) bounded loops with iteration counts governed by symbolic values, (4) local objective functions for data structures, and (5) global optimization criteria. Symbolic values make the sizes of arrays and other state flexible, allowing them to stretch as needed. Loops indexed by symbolic values make it possible to construct operations over elastic data structures. Objective functions provide a principled way for the programmer to describe the relative gain/loss from growing/shrinking individual data structures. Global optimization criteria make it possible to weight the relative importance of each structure or application residing on a shared device.

We have implemented a compiler for P4All that operates
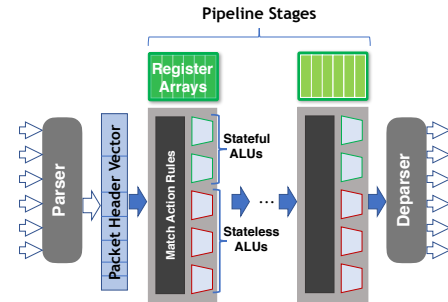
Figure 2: Protocol Independent Switch Architecture (PISA)

in two main stages. First, it computes an upper bound on the number of possible iterations of loops, so it can produce a simpler optimization problem over unrolled, loop-free code. This upper bound is computed by conservatively analyzing the dependency structure of the loop bodies and their resource utilization. Next, the compiler unrolls the loops to those bounds and generates a constraint system that optimizes the resource utilization of the loop-free code for a particular target. We use the Intel Tofino chip as our target. We evaluate our system by developing a number of reusable, elastic structures and building several elastic applications using these structures. Our experiments show that the P4All compiler runs in a matter of minutes (or less) and produces P4 programs that are competitive with hand-optimized code. This paper builds on our earlier workshop paper [21] by extending the language for nonlinear objective functions over multiple variables. We also implement the optimization problem and compiler outlined in the workshop paper, along with evaluating it with a variety of data structures.

In summary, we make the following contributions.

- The design of P4All, a backward-compatible extension to P4 that enables elastic network programming.

- The implementation of an optimizing compiler for P4All.

- A library of reusable elastic data structures, including their objective functions, and examples of combining them to create sophisticated applications.

- An evaluation of our system on a range of applications.

2 P4 Programming Challenges

Programming PISA devices is difficult because the resources available are limited and partitioned across pipeline stages. The architecture forces programmers to keep track of implicit dependencies between actions, lay out those actions across stages, compute memory requirements of each task, and fit the jigsaw pieces emerging from many independent tasks together into the overall resource-constrained puzzle of the pipeline.

2.1 Constrained Data-Plane Resources

P4 is designed to program a *Protocol Independent Switch Architecture* (PISA) data plane (Figure 2). Such an architecture contains a programmable packet parser, processing pipeline, and deparser. When a packet enters the switch, the parser extracts information from the packet and populates the *Packet Header Vector* (PHV). The PHV contains information from the packet's various fields, such as the source IP, TCP port, *etc.* that are relevant to the switch's task, whether it be routing, monitoring, or load balancing. The PHV also stores additional per-packet data, or *metadata*. Metadata often holds temporary values or intermediate results required by the application. Finally, the deparser reverses the function of the parser, using the PHV to reconstitute a packet and send it on its way.

Between parser and deparser sits a packet-processing pipeline. A program may recirculate a packet by sending it back to the beginning, but too much recirculation decreases throughput. Each stage contains a fixed set of resources.

- **Pipeline stages.** The processing pipeline is composed of a fixed number (S) of stages.

- **Packet header vector (PHV).** The PHV that carries information from packet fields and additional per-packet metadata through the pipeline has limited width (P bits).

- **Registers.** A stage is associated with M bits of registers (of limited width) that serve as persistent memory.

- **Match-action rules.** Each stage stores match-action rules in either TCAM or SRAM (T bits).

- **ALUs.** Actions are performed by ALUs associated with a stage. Each stage has F stateful ALUs (that perform actions requiring registers) and L stateless ALUs (that do not).

- **Hash units.** Each stage can perform N hashes at once.

The P4 language helps manage data-plane resources by providing a layer of abstraction above PISA. A P4 compiler maps these higher-level abstractions down to the PISA architecture and organizes the computation into stages. However, experience with programming in P4 suggests, that while a good start, the language is simply *not abstract enough*. It asks programmers to make fixed choices ahead of time about the size of data structures and the amount of computation the programmer believes the compiler can squeeze onto a particular PISA switch. To do this well, programmers must recognize dependencies between actions, estimate the stages available and consider the memory layout and usage of their programs—in short, they must redo many of the jobs of the compiler. These are difficult jobs to do well, even for world-experts, and next to impossible for novices. Inevitably, attempts at estimating resource bounds leads to some amount of trial and error. In summary, the current development environment requires a lot of fiddly, low-level work and takes human time and energy away from innovating at a high level of abstraction.

2.2 Example: Implementing NetCache in P4

To illustrate some of the difficulties of programming with P4, consider an engineer in charge of upgrading their network to include a new caching subsystem, based on NetCache [27], which is designed to accelerate response times for web services. NetCache contains two main data structures, a *count-min sketch* (CMS) for keeping track of the popularity of the keys, and a *key-value store* (KVS) to map popular keys to values. Like any good programmer, our engineer constructs these two data structures modularly, one at a time.

First, the engineer implements the CMS, a probabilistic data structure that uses multiple hash functions to keep approximate frequencies for a stream of items in sub-linear space. Intuitively, the CMS is a two-dimensional array of w columns and r rows. For each packet (x) that enters the switch, its flow ID (f_x) is hashed using r different hash functions ($\{h_i\}$), one for each row, that range from $(1 \ldots w)$. In each row, the output of the hash function determines which column in the row is incremented for f_x. For example, in the second row of the CMS, hash function h_2 determines that column ($h_2(f_x)$) is incremented. To approximate the number of times flow f_x has been seen, one computes the minimum of the values stored in columns $h_i(f_x)$ for all r rows.

The CMS may overestimate the number of occurrences of a packet x if there are hash collisions. Increasing the size of the sketch in any dimension—either by adding more rows (*i.e.*, additional, different hash functions) or by increasing the range of the hash functions—can improve accuracy. Our engineer must decide how to assign resources to the CMS, including how much memory to allocate and how to divide memory into rows. This allocation becomes even harder when grappling with dividing resources between multiple structures.

Figure 3 presents a fragment of a P4 program that implements a CMS. Lines 1-7 declare the metadata used by the CMS to store a count at a particular index (a hash of a flow id). Lines 10-12 declare the low-level data structures (registers) that actually make up the CMS—four rows ($r = 4$) of columns ($w = 2048$) that can each store values represented by 32 bits. Lines 14-16 and 18-20 declare the actions for hashing/incrementing and for updating the metadata designed to store the global minimum. Both actions use metadata, another constrained resource that must be accounted for. The hashing action is a complex action containing several atomic actions: (1) an action to hash the key to an index into a register array, (2) an action to increment the count found at the index, and (3) an action to write the result to metadata for use later in finding the global minimum. Such multi-part actions can demand a number of resources, including several ALUs. As our engineer adds more of these actions to the program, it becomes increasingly difficult to estimate the resource requirements. In the *apply* fragment of the P4 program (lines 22-30), the program first executes all the hash actions, computing and storing counts for each hash function, and then compares those counts

```
1  struct custom_metadata_t {
2      bit<32> min;
3      bit<32> index0;
4      bit<32> count0;
5      ...
6      bit<32> index3;
7      bit<32> count3; }
8  control Ingress( ... ) {
9      /* a register array for each hash table */
10     register<bit<32>>(2048) counter0;
11     ...
12     register<bit<32>>(2048) counter3;
13     /* an action to update each hash table */
14     action incr_0() { ... }
15     ...
16     action incr_3() { ... }
17     /* an action to set the minimum */
18     action min_0(){meta.min = meta.count0;}
19     ...
20     action min_3(){ . . . }
21     /* execute the following on each packet */
22     apply {
23         meta.min = 0; /*initialize global min*/
24         /* compute hashes */
25         incr_0(); ... incr_3();
26         /* compute minimum */
27         if (meta.count0 < meta.min) { min_0();}
28         ...
29         if (meta.count3 < meta.min) { min_3();}
30         } }
```

Figure 3: Count-Min Sketch in P4$_{16}$

to each other looking for the minimal one.

Upon reviewing this code, some of the deficiencies of P4 should immediately be apparent. First, there is a great deal of repeated code: Repeated data-structure definitions, action definitions, and invocations of those action definitions in the apply segment of the program. Good programming languages make it possible to avoid repeated code by allowing programmers to craft reusable abstractions. Avoiding repetition in programming has all sorts of good properties including the fact that when errors occur or when changes need to be made, they only need to be fixed/made in one place. Effective abstractions also help programmers change the number or nature of the repetitions easily. Unfortunately, P4 is missing such abstractions. One might also notice that the programmer had to choose magic constants (like 2048) and test whether such constants lead to programs that can be compiled or not.

3 Elastic Programming in P4All

P4All improves upon P4 by making it possible to construct and manipulate *elastic data structures*. These data structures may be developed modularly and combined, off-the-shelf, to build efficient new applications. In this section, we illustrate language features by building an elastic count-min sketch and using it in the NetCache application (see also Figure 4).

```
1  /* Count-min sketch module */
2  symbolic rows;
3  symbolic cols;
4  assume cols > 0;
5  assume 0 <= rows && rows < 4;
6  struct custom_metadata_t {
7      bit<32> min;
8      bit<32>[rows] index;
9      bit<32>[rows] count; }
10 register<bit<32>>(cols)[rows] cms;
11 action incr()[int index] { ... }
12 action min()[int index] { ... }
13 control hash_inc( ... ) {
14     apply {
15         for (i < rows) { incr()[i]; } } }
16 control find_min( ... ) {
17     apply {
18         for (i < rows) {
19             if (meta.count[i] < meta.min) {
20                 min()[i]; } } } }
21 objective cms_obj {
22     function: scale (3.0/cols);
23     step: 100; }
24
25 /* Key-value module */
26 symbolic k;  /* number of items */
27 assume k > 0;
28 control kv(...) {....}
29 /* NetCache module */
30 control NetCache( ... ) {
31     apply {
32         hash_inc.apply();
33         find_min.apply();
34         kv.apply(); } }
35 objective kvs_obj {
36     function: scale (sum(map(lambda y: 1.0/
                y,range(1,k+1))));
37     step: 100; }
38 maximize 0.8*kvs_obj-0.2*cms_obj
```

Figure 4: NetCache and Count-Min Sketch in P4All

3.1 Declare the Elastic Parameters

The first step in defining an elastic data structure is to declare the parameters that control the "stretch" of the structure. In the case of the count-min sketch there are two such parameters: (1) the number of rows in the sketch (*i.e.*, the number of hash functions), and (2) the number of columns (*i.e.*, the range of the hash). Such parameters are defined as *symbolic values*:

```
symbolic rows;
symbolic cols;
```

Symbolic integers like rows and cols should be thought of as "some integer"—they are placeholders that are determined (and optimized for) at compile time. In other words, as in other general-purpose, solver-aided languages like Boogie [29], Sketch [42], or Rosette [43], the programmer leaves the choice of value up to the P4All compiler.

Often, programmers know constraints that are unknown to the compiler. For instance, programmer experience might suggest that count-min sketches with more than four hash

functions offer diminishing returns. Such constraints may be written as assume statements as follows:

```
assume 0 <= rows && rows < 4;
```

An assume statement is related to the familiar assert statement found in languages like C. However, an assert statement *fails* (causing program termination) when its underlying condition evaluates to false. An assume statement, in contrast, always *succeeds*, but adds constraints to the system, guaranteeing the execution can depend upon the conditions assumed.

3.2 Declare Elastic State

The next step in defining an elastic data structure is to declare elastic state. P4 data structures are defined using a combination of the packet-header vector (metadata associated with each packet), registers (updated within the data plane), or match-action tables (rules installed by the control plane). The same is true of P4All. However, rather than using constants to define the extent of the state, one uses symbolic values, so the compiler can optimize their extents for the programmer.

In the count-min sketch, each row may be implemented as a register array (whose elements, in this case, are 32-bit integers used as counters). The number of registers in each register array is the number of columns in a row. In P4All, we define this matrix as a symbolic array of register arrays:

```
register<bit<32>>(cols)[rows] cms;
```

In this declaration, we have a symbolic array `cms`, which contains `rows` instances of the register type. Each register array holds `cols` instances of 32-bit values.

One can also define elastic metadata. For instance, for each row of the CMS, we need metadata to record an index and count for that row. To do so, we define symbolic arrays of metadata as follows. Each element of each array is a 32-bit field. The arrays each contain `rows` items.

```
bit<32>[rows] index;
bit<32>[rows] count;
```

3.3 Define Elastic Operations

Because elastic data structures can stretch or contract to fit available resources, elastic operations over those data structures must do more or less work in a corresponding fashion. To accommodate such variation, P4All extends P4 with loops whose iteration count may be controlled by symbolic values.

The count-min sketch of our running example consists of two operations. The first operation hashes the input `rows` times, incrementing the result found in the CMS at that location, and storing the result in the metadata. The second iterates over this metadata to compute the overall minimum found at all hash locations. Each operation is implemented using symbolic loops and is encapsulated in its own control block. The code below illustrates these operations.

```
/* actions used in control segments */
action incr()[int i] { ... }
action min()[int i] { ... }
/* hash and increment */
control hash_inc( ... ) {
  apply {
    for (i < rows) {
      incr()[i]; } } }
/* find global minimum */
control find_min( ... ) {
  apply {
    for (i < rows) {
      if (meta.count[i] < meta.min) {
        min()[i]; } } } }
```

These simple symbolic iterations (`for i < rows`) iterate from zero up to the symbolic bound (`rows`), incrementing the index by one each time. The overarching NetCache algorithm can now call each control block in the ingress pipeline.

```
control NetCache( ... ) {
  apply {
    hash_inc.apply(...);
    find_min.apply(...);
    ... } }
```

3.4 Specify the Objective Function

Data structures written for programmable switches are valid for a range of sizes. In the CMS example above, multiple assignments to `rows` and `cols` might fit within the resources of the switch. Finding the right parameters becomes even harder when a program has multiple data structures. In the case of NetCache, after defining a CMS, the programmer still needs to define and optimize a key-value store.

To automate the process of selecting parameters, P4All allows programmers to define an objective function that expresses the relationship between the utility of the structure and its size (as defined by symbolic values). For example, the CMS gains utility as one increases the cols parameter, because CMS error rate decreases. The P4All compiler should find instances of the symbolic values that optimize the given user-defined function subject to the constraint that the resulting program can fit within the switch resources.

For example, we can define the hit ratio for the key-value store as a function of its size for a workload with a Zipfian distribution. Suppose the key-value store has k items. The probability of a request to the ith most popular item is $\frac{1}{i^\alpha}$ [9]. In this case, α is a workload-dependent parameter that captures the amount of skew in the distribution. Then, for k items, the probability of a cache hit is the sum of the probabilities for each item in the key-value store: $\sum_{i=1}^{k} \frac{1}{i^\alpha}$. Hence, in P4All, for $\alpha = 1$, we might define the following objective function.

```
sum(map(lambda y: 1.0/y, range(1,k+1)))
```

In practice, we have found that non-linear optimization functions that use division can generate poor quality solutions, perhaps due to rounding errors (at least for the solver,

Gurobi [18], that we use). Hence, we *scale* such functions up, which results in the following optimization function.

```
scale(sum(map(lambda y: 1.0/y,range(1,k+1))))
```

Because we supply programmers with a library of reusable structures and optimization functions for them, non-expert programmers who use our libraries do not have to concern themselves with such details.

Similarly, we can define CMS error, ε, in terms of the number of columns, w, in the sketch. For a workload with parameter α, we can set $w = 3(1/\varepsilon)^{1/\alpha}$ [13]. The number of rows in the CMS does not affect ε, so we may choose to leave it out of the objective function. However, we can incorporate constraints to guarantee a minimum number of rows. The number of rows, d, in a CMS is used to determine a bound on the confidence, δ, of the estimations in the sketch ($d = 2.5 \ln 1/\delta$) [13]. For $\alpha = 1$, this objective function is `3.0/cols`.

In NetCache, the programmer must decide if either data structure should receive a higher proportion of the resources. If the CMS is prioritized, it can more accurately identify heavy hitters. However, the key-value store may not have sufficient space to store the frequently requested items. Conversely, if the CMS is too small, it cannot accurately measure which keys are popular and should be stored in the cache.

To capture the balance between data structures, a programmer can combine the objectives of each data structure into a weighted sum. For the NetCache application, this means creating an objective function that slightly prioritizes the hit rate of the key-value store over the error of the CMS:

```
maximize 0.8*kvs_obj−0.2*cms_obj
```

Figure 5 presents the symbolic values and possible objective functions for different data structures. Each structure has symbolic values and an objective function derived from the purpose of the structure, which may vary across applications. For example, the key-value store used in NetCache [27] acts as a cache, and the main goal of the algorithm is to maximize the cache hits. In the case of a collision in the hash table used in BeauCoup [10], only one of the values is kept , and the other is discarded, resulting in possible errors. Therefore, the main goal of the algorithm is to minimize collisions. The programmer can define the objective function of each structure based on the specific needs of the system. Existing analyses of common data structures can assist in defining these functions. For example, for the Bloom filter, the probability for false positives in Zipfian-distributed traffic has been analyzed by Cohen and Matias [12].

Complex Objectives. Some objective functions (*e.g.*, CMS) may only include a single symbolic variable, while others are a function of multiple variables (*e.g.*, Bloom filter in Figure 5). Because our compiler uses Gurobi [18] in the back end to solve optimization problems, it is bound by

Gurobi's constraints. In particular, Gurobi cannot solve complex, non-linear objectives that are functions of multiple variables directly. As a consequence, we tackle these objectives in two steps. First, we transform objectives in multiple variables (say, *x* and *y*) into objectives in a single variable (say *x*), by choosing a set of possible values of *y* to consider. We create a different Gurobi instance for each value of *y*, solve all the instances independently (a highly parallelizable task) and find the global optimum afterwards. Second, we use Gurobi to implement piece-wise linear approximations of the non-linear functions. Both of these steps benefit from some user input, and we have extended P4All to accommodate such input.

To reduce objectives with multiple variables to a single variable, we allow users to provide a set of points at which to consider evaluating certain symbolic values. Doing so provides users some control over the number of Gurobi instances generated and hence the compilation costs of solving complex optimization problems. Such sets can be generated via "range notation" (optionally including a stride, not shown here). For example, a possible objective function for a Bloom filter depends on the number of bits in the filter as well as the number of hash functions used. To eliminate the second variable from the subsequent optimization objective, a programmer can define the symbolic variable `hashes` as follows.

```
symbolic hashes [1..10]
```

On processing such a declaration, the compiler generates ten separate optimization problems, one for each potential value of the hash functions. The compiler chooses the solution from the instance that generated the optimal objective, and it outputs the program layout and the concrete values for the number of hashes and number of bits in the filter.

To reduce non-linear functions to linear ones, piecewise linear approximations are used. By default, the compiler will use the simplest such approximation: a single line. Doing so results in fast compile times, but can lead to suboptimal solutions. To improve the quality of solution, we allow programmers to specify the number of linear pieces using a "step" annotation on their objective function. For instance, on lines 21-23 of Figure 4, the objective for the CMS is defined with a simple function and a "step" of 100, indicating that a linear component is created between every 100th value. Increasing the number of linear components in the approximation can increase the cost of solving these optimization problems. By providing programmers with optional control, we support a "pay-as-you-go" model that allows programmers to trade compile time for precision if they so choose.

4 Compiling Elastic Programs

Inputs to the P4All compiler include a P4All program and a specification of the target's resources (*i.e.*, the PISA resource parameters defined Section 2.1 and the capabilities of the ALUs). The compiler outputs a P4 program with a concrete

Module	Symbolic values	Intuition	Objective Function
Key-value store/ hash table	Number of rows k	NetCache [27]: Maximize cache hits	maximize $\sum_{i=1}^{k} \frac{1}{i^\alpha}$
Hash-based matrix (Sketch)	Num rows d, num columns w	NetCache [27] (CMS): Minimize heavy hitter detection error	minimize $\varepsilon = (\frac{3}{w})^\alpha$
Bloom filter	Num bits m, num hash functions k	NetCache [27]: Minimize false positives. Expected number of items in stream n	minimize $(1 - e^{-\frac{kn}{m}})^k$
Multi-value table	Number of rows k	BeauCoup [10]: Minimize collisions. BeauCoup parameter set B; Probability to insert to table p = $f(\alpha, B)$; Expected number of items in stream n	minimize $(\frac{1}{k})^{n \cdot p}$
Sliding window sketch	Num rows d, num columns w, num epochs t	ConQuest [11]: Maximize epochs and minimize error	maximize $t(1 - (\frac{3}{w})^\alpha)$
Ring buffer	Buffer length b	Netseer [48]: Maximize buffer capacity	maximize b

Figure 5: Symbolic values and objective functions for Zipfian distributed traffic with (constant) parameter α.

assignment for each symbolic value, and a mapping of P4 program elements to stages in the target's pipeline. The output program is a *valid instance* of the input when the concrete values chosen to replace symbolic ones satisfy the user constraints (*i.e.*, `assume` statements) as well as the constraints of the PISA model that is targeted. In addition, loops are unrolled as indicated given the chosen concrete values. The output program is an *optimal instance*, when in addition to being valid, it maximizes the given objective function.

The P4All compiler first analyzes the control and data dependencies between actions in the program to compute an *upper bound* on the number of times each loop can be unrolled without exhausting the target's resources (§4.1). For example, a for-loop with a dependency across successive iterations cannot run more times than the number of pipeline stages (S). The unrolled program also cannot require more ALUs than exist on the target (($F + L$) * S).

Next, the compiler generates an integer linear program (ILP) with variables and constraints that govern the quantity and placement of actions, registers, and metadata relative to the target constraints (§4.2). The upper bound ensures this integer linear program is "large enough" to consider all possible placements of program elements that can maximize the use of resources. However, the ILP is more accurate than the coarse unrolling approximation we use. Hence, it may generate a solution that excludes some of the unrolled iterations—some of the later iterations may ultimately not "fit" in the data plane or may not optimize the user's preferred objective function when other constraints are accounted for. The resulting ILP solution is a layout of the program on the target, including the stage placement and memory allocation, and optimal concrete assignments for the symbolic values. Throughout this section, we use the CMS program in Figure 4 as a running example. For the sake of the example, we assume that the target has three pipeline stages ($S = 3$), 2048b memory per stage ($M = 2048$), two stateful and two stateless ALUs per stage ($F = L = 2$), and 4096 bits of PHV ($P = 4096$).
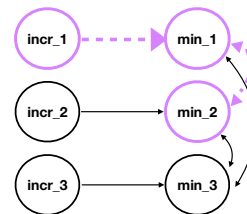
Figure 6: An example dependency graph used for computing upper bounds for loop unrolling (§4.1).

4.1 Upper Bounds for Loop Unrolling

In its first stage, the P4All compiler finds upper bounds for symbolic values bounding the input program's loops. To find an upper bound for a symbolic value v governing the number of iterations of some loop, the compiler first identifies all of the loops bounded by v. It then generates a graph G_v that captures the dependencies between the actions in each iteration of each loop and between successive iterations. It uses the information represented in G_v and the target's resource constraints to compute the upper bound.

Determining dependencies. When a loop is unrolled K times, it is replaced by K repetitions of the code in its body such that in repetition i, each action a in the original body of the loop is renamed to a_i. The compiler constructs the dependency graph G_v based on the actions in the unrolled bodies of for-loops bounded by v. Each node n in the dependency graph G_v represents a set A_n of actions that access the same register and thus *must* be placed in the same stage.

Dependency graphs can have (1) *precedence edges*, which are one-way, directed edges, and (2) *exclusion edges*, which are bidirectional. There is a precedence edge from node n_1 to node n_2 (indicated with the notation $n_1 \longrightarrow n_2$) if there is a data or control dependency from any of the actions represented by n_1 to any of the actions represented by n_2. The presence of the edge $n_1 \longrightarrow n_2$ forces all actions associated with n_1 to be placed in a stage that strictly precedes the stage where actions of n_2 are placed. In contrast, an exclusion edge ($n_1 \longleftrightarrow n_2$) indicates the actions of n_1 must be placed in a

separate stage from the actions of n_2 but n_1 need not precede n_2. In general, when actions are commutative, but cannot share a stage, they will be separated by exclusion edges. For instance, if actions a_1 and and a_2 both add one to the same metadata field, they cannot be placed in the same stage, but they commute: a_1 may precede a_2 or a_2 may precede a_1.

Figure 6 shows the dependency graph for `rows` from our CMS example. Only the `incr_i` actions access register arrays, and they all access different arrays. Thus, each node represents only one action. There is a precedence edge from `incr_i` to `min_i` as the former writes to the same metadata variable read by the latter. Thus, `incr_i` must be placed in a stage preceding `min_i`. There are exclusion edges between each pair of `min_i` and `min_j` because they are commutative but write to the same metadata fields: `min_i` sets the metadata variable tracking the global minimum `meta.min` to the minimum of its current value and the ith row of the CMS (`meta.count[i]`).

Computing the upper bound. To compute an upper bound for loops guarded by v, our compiler unrolls for-loops bounded by v for increasing values of K, generating a graph G_v until one of the following two criteria are satisfied:

1. the length of the longest simple path in G_v exceeds the total number of stages S, or

2. the total number of ALUs required to implement actions across all nodes in G_v exceeds the total number of ALUs on the target (*i.e.*, $(F+L)*S$).

Once either of the above criteria are satisfied, the compiler can use the current value of K, *i.e.*, the number of times the loops have been unrolled, as an upper bound for v. This is because any simple path in G_v represents a sequence of actions that must be laid out in disjoint stages. Hence, a simple path longer than the total number of stages cannot be implemented on the switch (*i.e.*, criteria 1). Likewise, the switch has only $(F+L)*S$ ALUs and a computation that requires more cannot be implemented (*i.e.*, criteria 2).

Figure 6 presents an analysis of a CMS loop bounded by `rows`. Notice that the length of the longest simple path in G_{rows} will exceed the number of stages ($S = 3$) when three iterations of the loop have been unrolled. On the other hand, when only two iterations of the loop are unrolled, the longest simple path has length 3 and will fit. Thus, the compiler computes 2 as the upper bound for this loop.

Nested loops. To manage nested loops, we apply the algorithm described above to each loop, making the most conservative assumption about the other loops. For instance, suppose the program has a loop with nesting depth 2 in which the outer loop bounded by v_{out} and the inner loop is bounded by v_{in}. Assume also the valid range of values for both v_{in} and v_{out} is $(1, \infty]$. The compiler sets v_{in} to one, unrolls the inner loop, and computes an upper bound for v_{out} as described above. Next, the compiler sets v_{out} to one, unrolls the outer loop, and proceeds to compute the upper bound for v_{in} as described

Variables		
Actions	#1	$\{x_{a_i,s} \mid 0 \le s < S\}$
Registers	#2	$\{m_{r_i,s} \mid 0 \le s < S\}$
Match-Action Tables	#3	$\{tm_{t_i,s} \mid 0 \le s < S\}$
Metadata	#4	$\{d_i \mid i \le U_v\}$
Constraints		
Dependencies		
Same-Stage	#5	$x_{a_i,s} = x_{b_i,s}\, s < S$
Exclusion	#6	$x_{a_i,s} \le 1 - x_{b_i,s}$
		$s < S$
Precedence	#7	$x_{b_i,y} \le 1 - x_{a_i,z}$
		$y, z < S, y \le z$
Conditional	#8	$\sum_{0 \le s < S} x_{a_i,s} = \sum_{0 \le s < S} x_{b_i,s}$
		$0 \le i \le U_v$
Resources		
Memory	#9	$\sum_i m_{r_i,s} \cdot w_{r_i} \le M \quad \forall s < S$
	#10	$m_{r_i,s} \le x_{a_i,s} \cdot M \quad 0 \le s < S$
	#11	$m_{r_i,s} \cdot w_0 = m_{0,s} \cdot w_{r_i}$
		$\forall s < S, r \ge 1$
TCAM	#12	$\sum_i tm_{t_i,s} \cdot tw_{t_i} \le T \quad \forall s < S$
Stateful ALUs	#13	$\sum_i H_f(a_i) \cdot x_{a_i,s} \le F$
		$\forall 0 \le s < S$
Stateless ALUs	#14	$\sum_i H_l(a_i) \cdot x_{a_i,s} \le L$
		$\forall 0 \le s < S$
PHV	#15	$\sum_i d_i \cdot bits_d \le P - P_{fixed}$
	#16	$d_i = \sum_{0 \le s < S} x_{a_i,s}$
		if accesses(a, d)
Hash Functions	#17	$\sum_i h_{ha_i,s} \le N \quad \forall s < S$
Others		
At Most Once	#18	$\sum_{0 \le s < S} x_{a_i,s} \le 1$
Inelastic Actions	#19	$\sum_{0 \le s < S} x_{a_{ne},s} = 1$

Figure 7: ILP Summary

above. In theory, heavily nested loops could lead to an explosion in the complexity of our algorithm, but in practice, we have not found nested loops common or problematic. Only our SketchLearn application requires nested loops and the nesting depth is just 2, which is easily handled by our system.

4.2 Optimizing Resource Constraints

After unrolling loops, the compiler has a loop-free program it can use to generate an integer linear program (ILP) to optimize. Figure 7 summarizes the ILP variables and constraints. Below, we use the notation #k to refer to the ILP constraint or variable labeled k in Figure 7.

Action Variables. To control placement of actions, the compiler generates a set of ILP variables named $x_{a_i,s}$ (#1). The variable $x_{a_i,s}$ is 1 when the action a_i appears in stage s of the pipeline and is 0 otherwise. For instance, in the count-min sketch, there are two actions (`incr` and `min`). If we unroll a loop containing those actions twice and there are three stages in the pipeline, we generate the following action variable set.

$$\{x_{a_i,s} \mid a \in \{\texttt{incr},\texttt{min}\}, \quad 1 \le i \le 2, \quad 0 \le s < S\}$$

Register Variables. In a PISA architecture, any register accessed by an action must be placed within the same stage. Thus placement (and size) of register arrays interact with placement of actions. For each register array r and pipeline

stage s, the ILP variable $m_{r,s}$ contains the amount of memory used to represent r in stage s (#2). This value will be zero in any stage that does not contain r and its associated actions. For instance, to allocate the cms registers, the compiler uses:

$$\{m_{\text{cms}_i,s} \mid 1 \le i \le 2, 0 \le s < S\}$$

Match-Action Table Variables. These variables represent the resources used by match-action tables. Similar to register variables, the variable $tm_{t_i,s}$ represents the amount of TCAM used by table t_i in stage s (#3). Note that in our current ILP, we assume that all tables, ones with and without ternary matches, use TCAM. We plan to extend the ILP so that it can choose to implement tables without ternary matches in SRAM.

Metadata Variables. The amount of metadata needed is also governed by symbolic values. If U_v is the upper bound on the symbolic value that governs the size of a metadata array, then the compiler generates a set of metadata variables d_i for $1 \le i \le U_v$ (#4). Each such variable will have value 1 in the ILP solution if that chunk of metadata is required and constraints described later will bound the total metadata to ensure it does not exceed the target size limits. In our running example, the bound U_v corresponds to the number of iterations of the loop that finds the global minimum value in the CMS.

Dependency Constraints. If a set of actions use the same register, they must be placed on the same stage. To do so, the compiler adds a *same-stage constraint* (#5). Similarly, if an action has a data or control dependency on another action, the two must be placed in separate stages. If there is an exclusion edge between actions a_i and b_i, the compiler creates a constraint to prevent these actions from being placed in the same stage (#6). If there is a precedence edge between actions a_i and b_i, the compiler creates a constraint forcing a_i to be placed in a stage before b_i (#7).

Conditional Constraints. In some cases, as it happens in our CMS example, multiple loops are governed by the same symbolic values. Hence, iterations of one loop (and the corresponding actions/metadata) exist if and only if the corresponding iterations of the other loop exist. Moreover, if any action within a loop iteration cannot fit in the data plane, then the entire loop iteration should not be instantiated at all. Conditional constraints (#8) enforce these invariants.

Resource Constraints. We generate ILP constraints for each of the resources listed in §2.1. Our ILP constraints reflect the memory limit per stage (#9) and the fact that memory and corresponding actions must be co-located (#10). The compiler also generates constraints to ensure that each register array in an array of register arrays has the same size (#11). Moreover, the ILP includes a constraint to guarantee that the TCAM tables in a stage fit within a stage's resources (#12).

To enforce limits on the number of stateful and stateless ALUs used in each stage, we assume that the target provides two functions $H_f(a_i)$ and $H_l(a_i)$ as part of the target specification. These functions specify the number of stateful and stateless ALUs, respectively, required to implement a given

action a_i on the target. Given that information, the compiler generates constraints to ensure that the total number of ALUs used by actions in the same stage do not exceed the available ALUs in a stage (#13, #14).

To track the use of PHV, constraint #15 ensures d_i is 1 whenever the action a_i (which accesses data d_i) is used in loop iteration i. To limit the total number of PHV bits, constraint #16 sums the size in bits ($bits_d$) of the metadata d associated with iteration i and enforces it to be within the PHV bits available to elastic program components ($P - P_{fixed}$, where P_{fixed} is the amount of metadata not present in elastic arrays). Finally, each stage in the PISA pipeline can perform a limited number of hash functions. To capture that, the compiler generates constraint #17, which ensures that the number of actions including a hash function h in each stage does not exceed the available number of available hashing units N.

Other Constraints. The compiler generates a constraint so that each action a_i is placed at most once (#18). Moreover, the compiler ensures that each *inelastic* action a_{ne} (*i.e.,* an action not encapsulated in a loop bounded by a symbolic value) must be placed in the pipeline (#19). Finally, any assume statements appearing in the P4All program are included in the ILP.

4.3 Limitations

Our current ILP formulation assumes each register array and match-action table can be placed in at most one stage. However, a PISA target could conceivably spread a single array or table across multiple pipeline stages. To accommodate multi-stage arrays or tables, we can relax the ILP constraint on placing actions in at most one stage (#18).

Moreover, some compilers further optimize the use of the PHV. For example, after a metadata field has been accessed, the PHV segment storing that field could be overwritten in later stages if the metadata were never accessed again. Our prototype does not yet capture PHV field reuse.

P4All optimizes with mostly static criteria. We do not consider any dynamic components, unless a programmer incorporates a workload-dependent parameter in their objective function. P4All also does not support elastic-width fields or parameterized packet recirculation. We leave these features, as well as PHV reuse, for future work.

5 Prototype P4All Compiler

In this section, we describe our prototype P4All compiler, written in Python.

Target specification. We created a target specification for the Intel Tofino switch, based on product documentation. The specification captures the parameters in Section 2.1 and the H_f and H_l functions that specify the number of ALUs required to implement a given action. Since the Tofino design is proprietary, our specification unquestionably omits some low-level constraints not described in the documentation; with

Applications	P4All Code	Compile Time (sec)	ILP (Var, Constr)
Linear Objective			
IPv4 Forwarding + Stateful Firewall	217	0.4	(192, 1026)
BeauCoup	541	0.1	(672,7511)
Precision	166	25.7	(1316, 18969)
NetChain	242	27.9	(252, 3278)
Elastic Switch.p4	804	0.2	(1080, 21581)
Non-Linear Objective			
Key-value store (KVS)	127	15.4	(168, 857)
Count-min sketch (CMS)	82	1.8	(396, 1994)
KVS + CMS (Section §3)	170	27.9	(586, 2815)
Non-Elastic Switch.p4 + CMS	853	17.5	(1498, 23575)
SketchLearn	445	2.4	(768, 880)
ConQuest	362	5.8	(612, 3734)
Multivariate Objective			
Bloom filter	70	513.6 (longest) 170.0 (avg)	(240, 308) (132, 191)
CMS + Bloom	223	67.3 (longest) 38.1 (avg)	(658, 2266) (550, 2149)

Figure 8: P4All applications, showing the lines of code in the P4All implementation. For structures with multiple instances, the last two columns give statistics for the single instance with the *longest* compile time and the *average* of all instances.

knowledge of such constraints, we could augment our target specification and optimization framework to handle them.

Compute upper bounds for symbolic values. To compute upper bounds and unroll loops, our prototype must analyze P4 dependencies. To facilitate this, we use the Lark toolkit [1] for parsing. We have also written a Python program that finds dependencies between actions and tables and outputs the information in a format our ILP can ingest. At the moment, we only produce precedence edges. As a result, we do not process exclusion edges, treating all edges as precedence edges. We plan to upgrade this in the future.

Generate and solve ILP. Our prototype generates the ILP with variables and constraints in Figure 7, as well as the objective function. We then invoke the Gurobi Optimizer [18] to compute a concrete assignment for each symbolic value. We then use these values to generate the unrolled P4 code.

P4 compiler. After the compiler converts the P4All program into a P4 program, we invoke the (black box) Tofino compiler to compile the P4 program for execution on the underlying Tofino switch. If our experiments initially fail to compile to the Tofino switch because of proprietary constraints, we adjust our target specification and added `assume` statements to further constrain the memory allocated to register arrays. Ideally, the P4All compiler would be embedded within a target-specific compiler to automatically incorporate the proprietary constraints, without our needing to infer them.
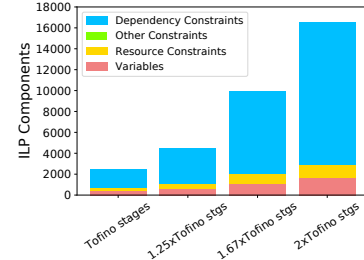
(a) Number of ILP variables and constraints for CMS as stages increase.

Num Stages	ILP Time (s)
Tofino	1.8
1.25xTofino	4.5
1.67xTofino	53.1
2xTofino	216.0

(b) ILP completion time for CMS as stages increase.

Figure 9: ILP performance as number of available stages increases.

6 Performance Evaluation

6.1 Compiler Performance

Figure 8 reports the sizes of the constraint systems, and the compile times, for benchmark applications when compiled against our Tofino resource specification. We choose applications with a variety of features, including elastic TCAM tables (Switch.p4), multivariate objectives (Bloom filter), elastic and non-elastic components (IPv4 forwarding and stateful firewall), and multiple elastic components (KVS and CMS, CMS and Bloom filter). In our experiments, we found that the choice of objective function greatly impacts performance. For example, a non-convex objective function results in a mixed integer program (MIP) instead of an ILP, which significantly increases solving time. On the other hand, our applications with linear objective functions (e.g., Switch.p4, BeauCoup) typically had smaller compile times. Additionally, increasing the step size for an objective (i.e., reducing the number of values provided to the ILP) decreases compile time.

For the data structures we evaluated with objective functions with multiple variables (*e.g.*, Bloom filter), our compiler created multiple instances of the optimization problem. We report the average compile time and the average number of ILP variables and constraints *for each instance*, along with the statistics for the largest instance. Our prototype compiler is not parallelized, but could easily be in the future, allowing us to solve many (possibly all) instances at the same time. Compile times of each ILP instance for the Bloom filter application range from roughly one second to 8.5 minutes.

Compile time increases as we increase the number of elastic elements in a P4All program. We evaluate ILP performance by observing the solving time as we increase the number of elastic elements in a program. Compilation for a single elastic sketch completed in about 10 seconds, while compilation for

(a) CMS error rate as memory increases.   **(b)** CMS and KVS sizes for different objectives.   **(c)** Bloom filter false positive vs. # hash functions.
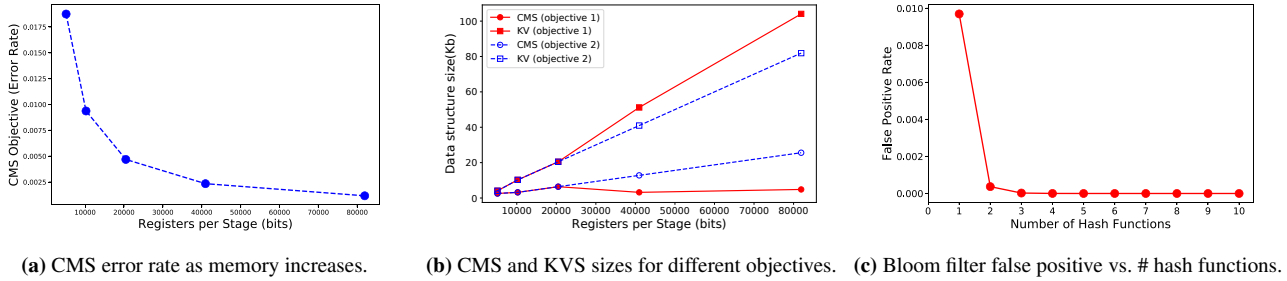
Figure 10: Elasticity of P4All

four sketches took over 30 minutes.

The number of constraints also affects compile time. The Bloom filter had the fewest ILP constraints, as it had no dependent components, and it alone had the largest compilation time. The reason for this is that the smaller number of constraints may lead to a more difficult optimization problem.

When we increase the available resources on the target, we generate a larger optimization problem, with more variables and constraints. Figure 9a the change in the number of constraints and variables as we increase the number of available stages on the target. Most of the resource and other constraints (*e.g.*, TCAM size, hash units, at most once, etc.) are linearly proportional to the stages. The dependency constraints are the only constraints that do not increase linearly with the stages. For a single P4All action, we create an ILP variable for each stage. However, the variables for CMS are not linearly proportional to the stages because as we increase stages, the upper bound on the actions also increases, resulting in more variables. Similarly, the ILP completion time increases super linearly with the number of stages (Figure 9b).

Some applications may have both elastic and non-elastic components. In our evaluations, we found that this did not significantly impact compile time. When we combined an elastic CMS and Switch.p4 (with fixed-size TCAM tables), the compile time was 17 seconds. Our compiler requires that all non-elastic portions of the program get placed on the switch, or the program will fail to compile.

Hand-written vs P4All-generated P4 To investigate whether P4All-generated P4 was competitive with hand-written P4, we examined a few P4 programs written by hand by other programmers and compared those programs with the P4 code generated from P4All. When we compare the number of registers used by the manually-written BeauCoup and the P4All-generated BeauCoup, we find they are exactly the same. ConQuest is made up of sketches, so we use the same objective function described in §3. With that function, our compiler tries to allocate as many registers as possible, and allocates all available space to sketches, as more registers means lower error. Examining the ConQuest paper in more depth, however, shows that the accuracy gains are minimal

after a certain point (2048 columns). To account for this, we easily adjust the objective function, and as a result, the compiled code uses exactly 2048 columns as in the original. This experiment illustrates the power of P4All beautifully. On one hand, our first optimization function is highly effective—it uses up all available resources. On the other hand, when new information arrives, like the fact that empirically, there are diminishing returns beyond a certain point, we need only adjust the objective function to reflect our new understanding of the utility. None of the implementation details need change. While this analysis is admittedly ad hoc, our findings here suggest that P4All does not put programmers at a disadvantage when it comes to producing resource-efficient P4.

6.2 Elasticity

In this section we measure how utility of data structures vary as resources are made available. Figure 10a shows how the error rate of a CMS decreases as we increase the available registers in each stage. Figure 10b shows how the sizes of a KVS and CMS change for different objective functions. We use the objective functions for KVS hit rate and CMS error rate as described in Figure 5. The first objective function $0.8 * (kv\_obj) - 0.2 * (cms\_obj)$ gives a higher weight to the KVS hit rate, while the second $0.2 * (kv\_obj) - 0.8 * (cms\_obj)$ gives a higher weight to the CMS error rate.

For multi-variate functions, the compiler generates multiple instances of the optimization problem, and chooses the solution to the instance with the best objective. In Figure 10c, we show the objective (false positive rate) from the instances of optimization for a Bloom filter. In each instance, the compiler increases the number of hashes used. The objective decreases for each instance, but not by much after the first instance.

6.3 Case Study

In a conversation with a major cloud provider, the researchers expressed interest in hosting a multiple applications on the same network device, which must include forwarding logic. We designed P4All for exactly such scenarios—elastic structures allow new applications to fit onto a shared device. We consider a simple case study oriented around this problem.

(a) Switch layout with forwarding tables and a CMS.
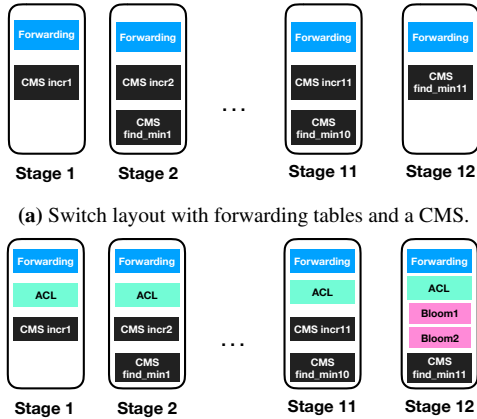
(b) Switch layout with forwarding tables, ACL tables, CMS, and Bloom filter used in stateful firewall.

Figure 11: Switch program layouts.

To do so, we started with the IPv4 forwarding code from switch.p4, but the size of the table is defined symbolically in P4All. We then added a CMS for heavy hitter detection. Figure 11a illustrates the layout: The forwarding tables utilize all of the TCAM resources, and the CMS uses registers.

Next, to demonstrate the flexibility and modularity of our framework, we add access control lists (ACLs), which use match-action tables, and squeeze in a stateful firewall, using Bloom filters, similar to the P4 tutorials [2]. Using P4, the programmer would manually resize the CMS and forwarding tables so the new applications could fit on the switch, but by using P4All, we do not have to change our existing code at all. To write ACLs with elastic TCAM tables, we modify the code in switch.p4 to include symbolic table sizes. Our compiler automatically resizes the elastic structures to fit on the switch, resulting in the layout in Figure 11b. The forwarding tables and ACLs now share the match-action table resources, and the registers in the Bloom filter fit alongside the CMS.

7 Related Work

Languages for network programming. There has been a large body of work on programming languages for software defined networks [3, 14, 37, 44] targeted towards OpenFlow [33], a predecessor to P4 [7, 36]. OpenFlow only allows for a fixed set of actions and not control over registers in the data plane, and so these abstractions are not sufficient for P4. While P4 makes it possible to create applications over a variety of hardware targets, it does not make it easy. Domino [40] and Chipmunk [16] use a high-level C-like language to aid in programming switches. P4All also aims to simplify this process, but we enhance P4 with elastic data structures. Domino and Chipmunk optimize the data-plane layout for static, fixed-sized data structures, and P4All optimizes the data structure itself to make the most effective use of resources.

Using synthesis for compiling to PISA. The Domino compiler extracts "codelets", groups of statements that must execute in the same stage. It then uses SKETCH [42] program synthesis to map a codelet to ALUs (atoms in the paper's terminology) in each stage. If any codelet violates target constraints, the program is rejected. To improve Domino, Chipmunk [16] uses syntax-guided synthesis to perform an exhaustive search of all mappings of the program to the target. Thus, it can find mappings that are sometimes missed by Domino. Lyra [15], extends this notion to a one-big-pipeline abstraction, allowing the composition of multiple algorithms to be placed across several heterogeneous ASICs. Nevertheless, Domino, Chipmunk and Lyra map programs with fixed-size data structures, while P4All enables elastic data structures.

Compiling to RMT. Jose et al. [28] use ILPs and greedy algorithms to compile programs for RMT [8] and FlexPipe [35] architectures. These ILPs are part of an all-or-nothing compiler which attempts to place actions on a switch based on the dependencies and the sizes of match-action tables. In contrast, the P4All compiler allows for elastic structures, which can stretch or compress according to a target's available resources.

Programmable Optimization. P^2GO [45] uses profile-guided optimization (*i.e.,* a sample traffic trace, not a static objective function) to reduce the resources required in a P4 program. P^2GO can effectively prune components that are not used in a given environment; however, if unexpected traffic turns up later, P^2GO may have pruned needed functionality!

8 Conclusion

In this paper, we introduce the concept of *elastic data structures* that can expand to use the resources on a hardware target. Elastic switch programs are more modular than their inelastic counterparts, as elastic pieces can adjust depending on the resource needs of other components on the switch. They also are portable, as they can be recompiled for different targets.

P4All is a backwards-compatible extension of P4 that includes symbolic values, arrays, loops and objective functions. We have developed P4All code for a number of reusable modules and several applications from the recent literature. We also implement and evaluate a compiler for P4All, demonstrating that compile times are reasonable and that auto-generated programs make efficient use of switch resources. We believe that P4All and our reusable modules will make it easier to implement and deploy a range of future data-plane applications.

Acknowledgments

References

[1] Lark parser. https://github.com/lark-parser/lark.

[2] Stateful firewall in P4. https://github.com/p4lang/tutorials/tree/master/exercises/firewall.

[3] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: Semantic foundations for networks. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 113–126. ACM, 2014.

[4] Ran Ben Basat, Xiaoqi Chen, Gil Einziger, Shir Landau Feibish, Danny Raz, and Minlan Yu. Routing oblivious measurement analytics. In *IFIP Networking*, pages 449–457, 2020.

[5] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minlan Yu, and Michael Mitzenmacher. PINT: Probabilistic in-band network telemetry. In *ACM SIGCOMM*, pages 662–680, 2020.

[6] Ran Ben Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. Efficient measurement on programmable switches using probabilistic recirculation. In *IEEE International Conference on Network Protocols*, pages 313–323, Sep. 2018.

[7] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[8] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *ACM SIGCOMM*, pages 99–110, 2013.

[9] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *IEEE INFOCOM*, pages 126–134, 1999.

[10] Xiaoqi Chen, Shir Landau Feibish, Mark Braverman, and Jennifer Rexford. BeauCoup: Answering many network traffic queries, one memory update at a time. In *ACM SIGCOMM*, pages 226–239, 2020.

[11] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, Ori Rottenstreich, Steven A. Monetti, and Tzuu-Yi Wang. Fine-grained queue measurement in the data plane. In *ACM SIGCOMM Conference on Emerging Networking EXperiments and Technologies*, pages 15–29. ACM, 2019.

[12] Saar Cohen and Yossi Matias. Spectral bloom filters. In *ACM SIGMOD*, pages 241–252. ACM, 2003.

[13] Graham Cormode and S. Muthukrishnan. Summarizing and mining skewed data streams. In Hillol Kargupta, Jaideep Srivastava, Chandrika Kamath, and Arnold Goodman, editors, *SIAM International Conference on Data Mining*, pages 44–55. SIAM, 2005.

[14] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. In *ACM SIGPLAN International Conference on Functional Programming*, pages 279–291. ACM, 2011.

[15] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. Lyra: A cross-platform language and compiler for data plane programming on heterogeneous ASICs. In *ACM SIGCOMM*, pages 435–450, 2020.

[16] Xiangyu Gao, Taegyun Kim, Michael D. Wong, Divya Raghunathan, Aatish Kishan Varma, Pravein Govindan Kannan, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. Switch code generation using program synthesis. In *ACM SIGCOMM*, page 44–61, 2020.

[17] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *ACM SIGCOMM*, pages 357–371. ACM, 2018.

[18] Gurobi Optimization. Gurobi optimizer reference manual. http://www.gurobi.com, 2019.

[19] Rob Harrison, Qizhe Cai, Arpit Gupta, and Jennifer Rexford. Network-wide heavy hitter detection with commodity switches. In *ACM Symposium on SDN Research*, 2018.

[20] Rob Harrison, Shir Landau Feibish, Arpit Gupta, Ross Teixeira, S. Muthukrishnan, and Jennifer Rexford. Carpe elephants: Seize the global heavy hitters. In *ACM SIGCOMM Workshop on Secure Programmable Network Infrastructure*, pages 15–21, 2020.

[21] Mary Hogan, Shir Landau-Feibish, Mina Tahmasbi Arashloo, Jennifer Rexford, David Walker, and Rob Harrison. Elastic switch programming with P4All. In *ACM Workshop on Hot Topics in Networks*, page 168–174, 2020.

[22] Thomas Holterbach, Edgar Costa Molero, Maria Apostolaki, Alberto Dainotti, Stefano Vissicchio, and Laurent Vanbever. Blink: Fast connectivity recovery entirely in the data plane. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 161–176, Boston, MA, February 2019.

[23] Qun Huang, Xin Jin, Patrick P. C. Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. SketchVisor: Robust network measurement for software packet processing. In *ACM SIGCOMM*, pages 113–126, 2017.

[24] Qun Huang, Patrick P. C. Lee, and Yungang Bao. Sketchlearn: Relieving user burdens in approximate measurement with automated statistical inference. In *ACM SIGCOMM*, pages 576–590, 2018.

[25] Theo Jepsen, Masoud Moshref, Antonio Carzaniga, Nate Foster, and Robert Soulé. Life in the fast lane: A line-rate linear road. In *ACM Symposium on SDN Research*, 2018.

[26] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. NetChain: Scale-free sub-RTT coordination. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 35–49, Renton, WA, April 2018.

[27] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soule, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing key-value stores with fast in-network caching. In *Symposium on Operating System Principles*, 2017.

[28] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. Compiling packet programs to reconfigurable switches. In *USENIX Conference on Networked Systems Design and Implementation*, pages 103–115, 2015.

[29] K. Rustan M. Leino and Philipp Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In Javier Esparza and Rupak Majumdar, editors, *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 312–327. Springer Berlin Heidelberg, 2010.

[30] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. FlowRadar: A better NetFlow for data centers. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 311–324, Santa Clara, CA, March 2016.

[31] Zaoxing Liu, Ran Ben Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. Nitrosketch: Robust and general sketch-based monitoring in software switches. In *ACM SIGCOMM*, pages 334–350, 2019.

[32] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with UnivMon. In *ACM SIGCOMM*, pages 101–114, 2016.

[33] Nick McKeown, Thomas E. Anderson, Hari Balakrishnan, Guru M. Parulkar, Larry L. Peterson, Jennifer Rexford, Scott Shenker, and Jonathan S. Turner. OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[34] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. SilkRoad: Making stateful layer-4 load balancing fast and cheap using switching ASICs. In *ACM SIGCOMM*, pages 15–28, 2017.

[35] Recep Ozdag. Intel® Ethernet Switch FM6000 Series–Software Defined Networking, 2012. `goo.gl/AnvOvX`.

[36] P4 Language Consortium. P4$_{16}$ language specifications, 2018. `https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.pdf`.

[37] Cole Schlesinger, Michael Greenberg, and David Walker. Concurrent NetCore: From policies to pipelines. In *ACM SIGPLAN International Conference on Functional programming*, pages 11–24, 2014.

[38] Naveen Kr. Sharma, Antoine Kaufmann, Thomas Anderson, Arvind Krishnamurthy, Jacob Nelson, and Simon Peter. Evaluating the power of flexible packet processing for network resource allocation. In *USENIX Networked Systems Design and Implementation*, pages 67–82, March 2017.

[39] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. Approximating fair queueing on reconfigurable switches. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 1–16, Renton, WA, April 2018.

[40] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In *ACM SIGCOMM*, pages 15–28, 2016.

[41] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In *ACM SIGCOMM Symposium on SDN Research*, pages 164–176, 2017.

[42] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *Architectural Support for*

Programming Languages and Operating Systems, pages 404–415, 2006.

[43] Emina Torlak and Rastislav Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 530–541, 2014.

[44] Andreas Voellmy, Junchang Wang, Y Richard Yang, Bryan Ford, and Paul Hudak. Maple: Simplifying SDN programming using algorithmic policies. In *ACM SIG-COMM*, volume 43, pages 87–98, August 2013.

[45] Patrick Wintermeyer, Maria Apostolaki, Alexander Diet-müller, and Laurent Vanbever. P2GO: P4 profile-guided optimizations. In *ACM Workshop on Hot Topics in Networks*, page 146–152, 2020.

[46] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *ACM SIGCOMM*, pages 561–575, 2018.

[47] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. NetLock: Fast, centralized lock management using programmable switches. In *ACM SIGCOMM*, pages 126–138, 2020.

[48] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, Pengcheng Zhang, Dennis Cai, Ming Zhang, and Mingwei Xu. Flow event telemetry on programmable data plane. In *ACM SIGCOMM*, pages 76–89, 2020.

Privid: Practical, Privacy-Preserving Video Analytics Queries

Frank Cangialosi⌂, Neil Agarwal ⚞, Venkat Arun⌂, Junchen Jiang◠, Srinivas Narayana⎊, Anand Sarwate⎊, Ravi Netravali ⚞

⌂ MIT CSAIL ⚞ Princeton University ◠ University of Chicago ⎊ Rutgers University

privid@csail.mit.edu

Abstract

Analytics on video recorded by cameras in public areas have the potential to fuel many exciting applications, but also pose the risk of intruding on individuals' privacy. Unfortunately, existing solutions fail to practically resolve this tension between utility and privacy, relying on perfect detection of all private information in each video frame—an elusive requirement. This paper presents: (1) a new notion of differential privacy (DP) for video analytics, (ρ, K, ϵ)-event-duration privacy, which protects all private information visible for less than a particular duration, rather than relying on perfect detections of that information, and (2) a practical system called PRIVID that enforces duration-based privacy even with the (untrusted) analyst-provided deep neural networks that are commonplace for video analytics today. Across a variety of videos and queries, we show that PRIVID increases error by 1-21% relative to a non-private system.

1 Introduction

High-resolution video cameras are now pervasive in public settings [1, 3–5, 10], with deployments throughout city streets, in our doctor's offices and schools, and in the places we shop, eat, or work. Traditionally, these cameras were monitored manually, if at all, and used for security purposes, such as providing evidence for a crime or locating a missing person. However, steady advances in computer vision [32, 51, 53, 55, 65] have made it possible to automate video-content analytics (both live and retrospective) at a massive scale across entire networks of cameras. While these trends enable a variety of important applications [2, 11, 13, 14] and fuel much work in the systems community [26, 30, 40, 43, 44, 47, 48, 54, 73], they also enable privacy intrusions at an unprecedented level [7, 64].

As a concrete example, consider the operator for a network of city-owned cameras. Different organizations (i.e., "analysts") want access to the camera feeds for a range of needs: (1) health officials want to measure the fraction of people wearing masks and following COVID-19 social distancing orders [38], (2) the transportation department wants to monitor the density and flow of vehicles, bikes, and pedestrians to determine where to add sidewalks and bike lanes [21], and (3) businesses are willing to pay the city to understand shopping behaviors for better planning of promotions [19].

Unfortunately, freely sharing the video with these parties may enable them to violate the privacy of individuals in the scene by tracking where they are, and when. For example, the "local business" may actually be a bank or insurance company that wants to track individuals' private lives for their risk models, while well-known companies [17] or government agencies may succumb to mission creep [18, 20]. Further, any organiza-

tions with good intentions could have employees with malicious intent who wish to spy on a friend or co-worker [15, 16].

There is an *inherent tension between utility and privacy*. In this paper, we ask: is it possible to enable these (untrusted) organizations to use the collected video for analytics, while also guaranteeing citizens that their privacy will be protected? Currently, the answer is no. As a consequence, many cities have outright banned analytics on public videos, even for law enforcement purposes [9, 12].

While a wide variety of solutions have been proposed (§3), ranging from computer vision (CV)-based obfuscation [23, 60, 68, 70] (e.g., blurring faces) to differential privacy (DP)-based methods [66, 67], they all use some variant of the same strategy: find *all* private information in the video, then hide it. Unfortunately, the first step alone can be unrealistic in practice (§3.1); it requires: (1) an explicit specification of all private information that could be used to identify an individual (e.g., their backpack), and then (2) the ability to spatially *locate* all of that information in *every* frame of the video—a near impossible task even with state-of-the-art CV algorithms [6]. Further, if these approaches cannot find some private information, they fundamentally cannot *know* that they missed it. Taken together, they can provide, at best, a conditional and brittle privacy guarantee such as the following: if an individual is only identifiable by their face, and their face is detectable in every frame of the video by the implementation's specific CV model in the specific conditions of this video, then their privacy will be protected.

This paper takes a pragmatic stance and aims to provide a definitively achievable privacy guarantee that captures the aspiration of prior approaches (i.e., individuals cannot be identified in any frame or tracked across frames) despite the limitations that plague them. To do this, we leverage two key observations: (1) a large body of video analytics queries are aggregations [47, 49], and (2) they typically aggregate over durations of video (e.g., hours or days) that far exceed the duration of any one individual in the scene (e.g., seconds or minutes) [47]. Building on these observations, we make three contributions by jointly designing a new notion of duration-based privacy for video analytics, a system implementation to realize it, and a series of optimizations to improve utility.

Duration-based differential privacy. To remove the dependence on spatially locating all private information in each video frame, we reframe the approach to privacy to instead focus on the temporal aspect of private information in video data, i.e., *how long* something is visible to a camera. More specifically, building on the differential privacy (DP) framework [37], we propose a new notion of privacy for

video, (ρ, K, ϵ)-event-duration privacy (formalized in §4.1): *anything* visible to a camera less than K times for less than ρ seconds each time ("(ρ, K)-bounded") is protected with ϵ-DP. The video owner expresses their privacy policy using (ρ, K), which we argue is powerful enough to capture many practical privacy goals. For example, if they choose $\rho = 5\text{min}$, anyone visible for less than 5 minutes is protected with ϵ-DP, which in turn prevents tracking them. We discuss other policies in §4.2.

This notion of privacy has three benefits. First, it decouples the definition of privacy from its enforcement. The enforcement mechanism does not need to make any decisions about what is private or find private information to protect it; everything (private or not) captured by the bound is protected. Second, a (ρ, K) bound that captures a set of individuals implicitly captures and thus protects any information visible for the same (or less) time without specifying it (e.g., an individual's backpack, or even their gait). Third, protecting all individuals in a video scene requires only their maximum duration, and estimating this value is far more robust to the imperfections of CV algorithms than precisely locating those individuals and their associated objects in each frame. For example, even if a CV algorithm misses individuals in some frames (or entirely), it can still capture a representative sample and piece together trajectories well enough to estimate their duration (§4.2).

Privid: a differentially-private video analytics system. Realizing (ρ, K, ϵ)-privacy (or more generally, any DP mechanism) in today's video analytics pipelines faces several challenges. In traditional database settings, implementing DP requires adding random noise proportional to the *sensitivity* of a query, i.e., the maximum amount that any one piece of private information could impact the query output. However, bounding the sensitivity is difficult in video analytics pipelines because (1) pipelines typically operate as bring-your-own-query-implementation to support the wide-ranging applications described earlier [22, 25, 26, 28, 29, 39, 41], and (2) these implementations involve video processing algorithms that increasingly rely on deep neural networks (DNNs), which are notoriously hard to inspect or vet (and thus, trust).

To bound the sensitivity necessary for (ρ, K, ϵ)-privacy while supporting "black-box" analyst-provided query implementations (including DNNs), Privid only accepts analyst queries structured in the following *split-process-aggregate* format (§5.2): (i) videos are split into temporally-contiguous chunks, (ii) each chunk of video is processed by an arbitrary analyst-provided processing program to produce an (untrusted) table, (iii) values in the table are aggregated (e.g. averaged) to compute a result, and (iv) noise is added to the result before release. The key in this pipeline is step (ii): we treat the analyst-provided program as an arbitrary Turing machine with restricted inputs (a single chunk of video frames and some metadata) and restricted outputs (rows of a table). As a result, only one chunk can contribute to the value of each row, and we know which chunk generated each row. If an individual is (ρ, K)-bounded, the number of chunks they appear

in is bounded, and thus the number of rows their presence can affect is bounded as well. With a bound on the number of rows, we can apply classic differential privacy techniques (§5.5).

Optimizations for improved utility. To further enhance utility, Privid provides two video-specific optimizations to lower the required noise while preserving an equivalent level of privacy: (i) the ability to mask regions of the video frame, (ii) the ability to split frames spatially into different regions, and aggregate results from these regions. These optimizations result in limiting the portion of the aggregate result that any individual's presence can impact, enabling a "tighter" (ρ, K) bound and in turn a higher quality query result.

Evaluation. We evaluate Privid using a variety of public videos and a diverse range of queries inspired by recent work in this space. Privid increases error by 1-21% relative to a non-private system, while satisfying an instantiation of (ρ, K, ϵ)-privacy that protects all individuals in the video. We discuss ethics in §9. Source code and datasets for Privid are available at https://github.com/fcangialosi/privid.

2 Problem Statement

2.1 Video Analytics Background

Video analytics pipelines are employed to answer high-level questions about segments of video captured from one or more cameras and across a variety of time ranges. Example questions include "how many people entered store X each hour?" or "which roads suffered from the most accidents in 2020?" (see §7.2 and Table 3 for more specific examples). A question is expressed as a *query*, which encompasses all of the computation necessary to answer that question.[1] For example, to answer the question "what is the average speed of red cars traveling along road Y?", the "query" would include an object detection algorithm to recognize cars, an object tracking algorithm to group them into trajectories, an algorithm for computing speed from a trajectory, and logic to filter only the red cars and average their speeds.

2.2 Problem Definition

Video analytics pipelines broadly involve four logical roles (though any combination may pertain to the same entity):

- **Individuals**, whose behavior and activity are observed by the camera.
- **Video Owner (VO)**, who operates the camera and thus owns the video data it captures.
- **Analyst**, who wishes to run queries over the video.
- **Compute Provider**, who executes the analyst's query.

In this work, we are concerned with the dilemma of a VO. The VO would like to enable a variety of (untrusted) analysts to answer questions about its videos (such as those in §2.1), as long as the results do not infringe on the privacy of the individuals who appear in the videos. Informally, privacy

---

[1] Our definition is distinct from related work, which defines a query as returning intermediate results (e.g., bounding boxes) rather than the final answer to the high-level question.

"leakage" occurs when an analyst can learn something about a specific individual that they did not know before executing a query. To practically achieve these properties, a system must meet three concrete goals:

1. **Formal notion of privacy**. The system's privacy policies should formally describe the type and amount of privacy that could be lost through a query. Given a privacy policy, the system should be able to provide a *guarantee* that it will be enforced, regardless of properties of the data or query implementation.

2. **Maximize utility for analysts.** The system should support queries whose final *result* does not infringe on the privacy of any individuals. Further, if accuracy loss is introduced to achieve privacy for a given query, it should be possible to bound that loss (relative to running the same query over the original video, without any privacy preserving mechanisms). Without such a bound, analysts would be unable to rely on any provided results.

3. **"Bring Your Own Model"**. Computer vision models are at the heart of modern video processing. However, there is not one or even a discrete set of models for all tasks and videos. Even the same task may require different models, parameters, or post-processing steps when applied to different videos. In many cases, analysts will want to use models that they trained themselves, especially when training involves proprietary data. Thus, a system must allow analysts to provide their own video-processing models.

It is important to note that the class of analytics queries we seek to enable are distinct from *security-oriented* queries (e.g., finding a stolen car or missing child), which *require* identification of a particular individual, and are thus directly at odds with individual privacy. In contrast, analytics queries involve searching for patterns and trends in large amounts of data; intermediate steps may operate over the data of specific individuals, but they do not distinguish individuals in their final aggregated result (§2.1).

2.3 Threat Model

The VO employs a privacy-preserving system to handle queries about a set of cameras it manages; the system retains full control over the video data, analysts can only interact with it via the query interface. The VO does not trust the analysts (or their query implementation code). Any number of analysts may be malicious and may collude to violate the privacy of the same individual. However, analysts trust the VO to be honest. Analysts are also willing to share their query implementation (so that the VO can execute it). The VO views this code as an untrusted blackbox which it cannot vet.

Analysts pose queries adaptively (i.e., the full set of queries is not known ahead of time, and analysts may utilize the results of prior queries when posing a new one). A single query may operate over video from multiple cameras. We assume the VO has sufficient computing resources to execute the query, either via resources that they own, or through the secure use of third-party resources [62].

The system releases some per-camera metadata publicly (§8.1), including a sample video clip. The resulting leak is interpretable and can be minimized by the VO. The system protects all other information with a formal guarantee of (ρ, K, ϵ)-privacy (Def 4.3).

3 Limitations of Related Work

Before presenting our solution, we consider prior privacy-preserving mechanisms (both for video and in general). Unfortunately, each fails to satisfy at least one of the goals in §2.2.

3.1 Denaturing

The predominant approach to privacy preservation with video data is *denaturing* [23, 34, 60, 68, 70, 72], whereby systems aim to obscure (e.g., via blurring [23] or blocking [68] as in Fig. 1) any private information in the video before releasing it for analysis. In principle, if nothing private is left in the video, then privacy concerns are eliminated.

The fundamental issue is that denaturing approaches require *perfectly* accurate and comprehensive knowledge of the spatial locations of private information in *every frame* of a video. Any private object that goes undetected, even in just a single frame, will not be obscured and thus directly leads to a leakage of private information.

To detect private information, one must first semantically define *what* is private, i.e., what is the full set of information linked, directly or indirectly, to the privacy of each individual? While some information is obviously linked (e.g., an individual's face), it is difficult to determine *all* such information for all individuals in all scenarios. For instance, a malicious analyst may have prior information that a VO does not, such as knowledge that a particular individual carries a specific bag or rides a unique bike (e.g., Fig. 1-B). Further, even with a semantic definition, detecting private information is difficult. State-of-the-art computer vision algorithms commonly miss objects or produce erroneous classification labels in favorable video conditions [74]; performance steeply degrades in more challenging conditions such as poor lighting, distant objects, and low resolution, all of which are common in public video. Taken together, the problem is that denaturing systems cannot guarantee whether or not a private object was left in the video, and thus fail to provide a formal notion of privacy (violating Goal 1).

Denaturing also falls short from the analyst's perspective. First, it inherently precludes (safe) queries that aggregate over private information (violating Goal 2). For example, an urban planner may wish to count the number of people that walk in front of camera A and then camera B. Doing so requires identifying and cross-referencing individuals between the cameras (which is not possible if they have been denatured), but the ag-
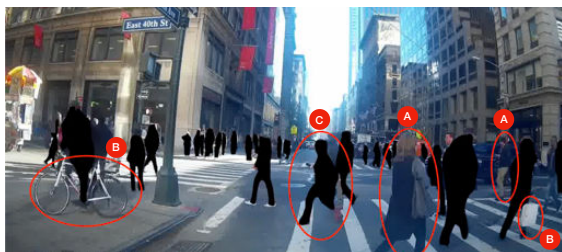
Figure 1: A video clip after (silhouette) denaturing exemplifying some of its shortcomings: (A) entirely missed detections, (B) potentially-identifying objects not incorporated in privacy definition, (C) silhouette may reveal gait.

gregate count may be large and safe to release.[2] Second, obfuscated objects are not naturally occurring and thus video processing pipelines are not designed to handle them. If the analyst's processing code and models have not been trained explicitly on the type of obfuscation the VO is employing, it may behave in unpredictable and unbounded ways (violating Goal 2).

3.2 Differential Privacy

Differential Privacy (DP) is a strong formal definition of privacy for traditional databases [37]. It enables analysts to compute aggregate statistics over a database, while protecting the presence of any individual entry in the database. DP is not a privacy-preserving mechanism itself, but rather a goal that an algorithm can aim to satisfy. Informally speaking, an algorithm satisfies DP if adding or removing an individual from the input database does not noticeably change the output of computation, almost as if any given individual were not present in the first place. More precisely,

DEFINITION 3.1. Two databases D and D' are *neighboring* if they differ in the data of only a single user (typically, a single row in a table).

DEFINITION 3.2. A randomized algorithm $\mathcal{A}$ is ϵ-*differentially private* if, for all pairs of neighboring databases (D, D') and all $S \subseteq Range(\mathcal{A})$:

$$\Pr[\mathcal{A}(D) \in S] \leq e^\epsilon \Pr[\mathcal{A}(D') \in S] \qquad (3.1)$$

A non-private computation (e.g., computing a sum of bank balances) is typically made differentially private by adding random noise sampled from a Laplace distribution to the final result of the computation [37]. The scale of noise is set proportional to the *sensitivity* (Δ) of the computation, or the maximum amount by which the computation's output could change due to the presence/absence of any one individual. For instance, suppose a database contains a value $v_i \in V$ for each user i, where $l \leq v_i \leq u$. If a query seeks to sum all values in V, any one individual's v_i can influence that sum by at most $\Delta = u - l$, and thus adding noise with scale $u - l$ would satisfy DP.

Challenges. Determining the sensitivity of a computation is the key ingredient of satisfying DP. It requires understanding

(a) how individuals are delineated in the data, and (b) how the aggregation incorporates information about each individual. In the tabular data structures that DP was designed for, these are straightforward. Each row (or a set of rows sharing a unique key) typically represents one individual, and queries are expressed in relational algebra, which describes exactly how it aggregates over these rows. However, these answers do not translate to video data; we next discuss the challenges in the context of several applications of DP to video analytics.

Regarding *requirement (a)*, as described in §3.1, it is difficult and error-prone to determine the full set of pixels in a video that correspond to each user (including all potentially identifying objects). Accordingly, prior attempts of applying DP concepts to video analytics [66, 67] that rely on perfectly defined and detected private information (via CV) fall short in the same way as denaturing approaches (violating Goal 1).

Regarding *requirement (b)*, typical video processing algorithms (e.g., ML-based CV models) are not transparent about how they incorporate private objects into their results. Thus, without a specific query interface, the "tightest" possible bound on the sensitivity of an arbitrary computation over a video is simply the entire range of the output space. In this case, satisfying DP would add noise greater than or equal to any possible output, precluding any utility (violating Goal 2).

Given that DP is well understood for tables, a natural idea would be for the VO to use their own (trusted) model to first convert the video into a table (e.g., of objects in the video), then provide a DP interface over *that table*[3] (instead of directly over the video itself). However, in order to provide a guarantee of privacy, the VO would need to completely trust the model that creates the table. This entirely precludes using a model created by the *untrusted* analyst (violating Goal 3).

4 Event Duration Privacy

We will first formalize (ρ, K, ϵ)-privacy, then provide the intuition for what it protects and clarify its limitations.

4.1 Definition

We consider a video V to be an arbitrarily long sequence of frames, sampled at f frames per second, recorded directly from a camera (i.e., unedited). A "segment" $v \subset V$ of video is a contiguous subsequence of those frames. The "duration" of a segment $d(v)$ is measured in real time (seconds), as opposed to frames. An "event" e is abstractly *anything* that is visible within the camera's field of view.

As a running example, consider a video segment v in which individual x is visible for 30 seconds before they enter a building, and then another 10 seconds when they leave some time later. The "event" of x's visit is comprised of one 30-second segment, and another 10-second segment.

---

[2]As a workaround, the VO could annotate denatured objects with query-specific information, but this would conflict with Goal 3.

[3]This would be equivalent to adding DP to an existing video analytics interface, such as [30, 47], which treat the video as a table of objects.

Definition 4.1 ((ρ, K)-bounded events). An event e is (ρ, K)-bounded if there exists a set of $\leq K$ video segments that completely contain[4] the event, and each of these segments individually have duration $\leq \rho$.

(Ex). The tightest bound on x's visit is ($\rho = 30s, K = 2$). To be explicit, x's visit is also (ρ, K)-bounded for any $\rho \geq 30s$ and $K \geq 2$.

Definition 4.2 ((ρ, K)-neighboring videos). Two video segments v, v' are (ρ, K)-neighboring if the set of frames in which they differ is (ρ, K)-bounded.

(Ex). One potential v' is a hypothetical video in which x was never present (but everything else observed in v remained the same). Note this is purely to denote the strength of the guarantee in the following definition, the VO does not actually construct such a v'.

Definition 4.3 ((ρ, K, ϵ)-event-duration privacy). A randomized mechanism $\mathcal{M}$ satisfies (ρ, K, ϵ)-event-duration privacy [5] iff for all possible pairs of (ρ, K)-neighboring videos v, v', any finite set of queries $Q = \{q_1, q_2, ...\}$ and all $S_q \subseteq Range(\mathcal{M}(\cdot, q))$:

$$Pr[(\mathcal{M}(v, q_1), ..., \mathcal{M}(v, q_n)) \in S_{q_1} \times \cdots \times S_{q_n}] \leq$$
$$e^\epsilon Pr[(\mathcal{M}(v', q_1), ..., \mathcal{M}(v', q_n))) \in S_{q_1} \times \cdots \times S_{q_n}]$$

Guarantee. (ρ, K, ϵ)-privacy protects all (ρ, K)-bounded events (such as x's visit to the building) with ϵ-DP: informally, if an event is (ρ, K)-bounded, an adversary cannot increase their knowledge of whether or not the event happened by observing a query result from $\mathcal{M}$. To be clear, (ρ, K, ϵ)-privacy is *not* a departure from DP, but rather an extension to explicitly specify what to protect in the context of video.

4.2 Choosing a Privacy Policy

The VO is responsible for choosing the parameter values (ρ, K) ("policy") that bound the class of events they wish to protect. They may use domain knowledge, employ CV algorithms to analyze durations in past video from the camera, or a mix of both. Regardless, they express their goal to PRIVID solely through their choice of (ρ, K).

Automatic setting of (ρ, K). The primary reason (ρ, K, ϵ)-privacy is *practical* is that, despite their imperfections, today's CV algorithms are capable of producing good estimates of the maximum duration any individuals are visible in a scene. We provide some evidence of this intuition over three representative videos from our evaluation. For each video, we chose a 10-minute segment and manually annotate the duration of each individual (person or vehicle), i.e., "Ground Truth", then use

Figure 2: The results of a state-of-the-art object detection algorithm (filtered to "person" class) on one frame of urban. The algorithm misses 76% of individuals in the frame, but is *still* able to produce a conservative bound on the maximum duration of all individuals (Table 1).

| Video | Maximum Duration | | % Objects |
	Ground Truth	CV Estimate	CV Missed
campus	81 sec	83 sec	29%
highway*	316 sec	439 sec	5%
urban	270 sec	354 sec	76%

Table 1: Despite the imperfection of current CV algorithms (exemplified by % objects they failed to detect), they still produce a conservative estimate on the duration of any individual's presence. *For the purposes of this experiment, we ignored cars that were parked for the entire duration of the segment.

state-of-the-art object detection and tracking to estimate the durations and report the maximum ("CV"). Our results, summarized in Table 1, show that, while object detection misses a non-trivial fraction of bounding boxes, the tracking algorithm is able to fill in the gaps for enough trajectories to capture a conservative estimate of the maximum duration. In other words, for our three videos, using these algorithms to parameterize a (ρ, K, ϵ)-private system would successfully capture the duration of, and thus protect the privacy of, *all* individuals, while using them to implement any prior approach would not.

Relaxing the set of private individuals. Sometimes protecting *all* individuals is unnecessary. Consider a camera in a store; employees will appear significantly longer and more frequently than customers (e.g., 8 hours every day vs. 30 minutes once a week), but if the fact that the employees work there is public knowledge, the VO can pick a policy (with smaller ρ and K) that only bounds the appearance of customers.

Generic policies. Alternatively, the VO can choose a policy to place a generic limit on the (temporal) granularity of queries. Consider a policy ($\rho = 5\text{min}, K = 1$). Suppose individual x stops and talks to a few people on their way to work each morning, but each conversation lasts less than 5 minutes. Although the policy does not protect x's presence or even the fact that they often stop to chat on their way to work, it *does* protect the timing and contents of each conversation.

4.3 Privacy Guarantees in Practice

In PRIVID's implementation of (ρ, K, ϵ)-privacy (described in the following section), the policy provides a relative reference point: events that exactly match the policy (i.e., made up of *exactly* K segments each of duration ρ) are protected

---

[4] A set of segments is said to completely contain an event if the event is not visible in any frames outside of those segments.

[5] We chose to use ϵ-DP rather than the more general (ϵ, δ)-DP for simplicity, since the difference is not significant to our definition. Our definition could be extended to (ϵ, δ)-DP without additional insights.

with ϵ-DP, while events that are visible for shorter or longer durations are protected with a proportionally (w.r.t. the duration) stronger or weaker guarantee, respectively.

Theorem 4.1. Consider a camera with a fixed policy (ρ, K, ϵ). If an individual x's appearance in front of the camera is bound by some $(\hat{\rho}, \hat{K})$, then PRIVID effectively protects x with $\hat{\epsilon}$-DP, where $\hat{\epsilon}$ is $O(\frac{\hat{\rho}\hat{K}}{\rho K})\epsilon$, which grows (degrades) as $(\hat{\rho}, \hat{K})$ increase while (ρ, K, ϵ) are fixed, and the constants do not depend on the query. We provide a formal proof in §A.1.

For example, given $(\rho = 1hr, K = 1)$, PRIVID would protect an a single 2-hour appearance with $\sim 2\epsilon$-DP (weaker) or a single half-hour appearance with $\sim \frac{1}{2}\epsilon$-DP (stronger).

Graceful degradation. An important corollary of this theorem is that privacy degrades "gracefully". As an event's $\hat{\rho}$ increases further from ρ (or $\hat{K}$ from K), its effective $\hat{\epsilon}$ increases linearly, yielding a progressively weaker guarantee. (The reverse is true, as $\hat{\rho}$ and $\hat{K}$ decrease, it yields a stronger guarantee). Thus, if $\hat{\rho}$ (or $\hat{K}$) is only *marginally* greater than ρ (or K), then the event is not immediately revealed in the clear, but rather is protected with $\hat{\epsilon}$-DP, which is still a DP guarantee, only marginally weaker: a malicious analyst has only a marginally higher probability of detecting x in the worst case. This in effect *relaxes* the requirement that (ρ, K) be set strictly to the maximum duration an individual could appear in the video to achieve useful levels of privacy. We generalize and provide a visualization of this degradation in §A.2.

Repeated appearances. The larger the time window of video a query analyzes, the more instances an individual may appear within the window, even if each appearance is itself bounded by ρ. Consider our example individual x and policy $(\rho = 30s, K = 2)$ from §4.1. In the query window of a single day d, x appears twice; they are properly (ρ, K)-bounded and thus the event "x appeared on day d" is protected with ϵ-DP. Now, consider a query window of one week; x appears 14 times (2 times per day), so the event "x appeared sometime this week" is $(\rho, 7K)$-bounded and thus protected with (weaker) 7ϵ-DP. However, the more specific event "x appeared on day d" (for any d in the week) is *still* (ρ, K)-bounded, and thus still protected with the same ϵ-DP. In other words, while an analyst may learn that an individual appeared *sometime* in a given week, they cannot learn on which day they appeared. Thus, in order to get greater certainty, the analyst must give up temporal granularity.

Multiple cameras. When an individual appears in front of multiple cameras, their privacy guarantees are analogous to the previous case of repeated appearances in a single camera. If they appear in front of N different cameras, where the event of their appearance in camera i is protected with $\hat{\epsilon}_i$-DP, then the event of their appearance across all the cameras is protected with $\sum_i \hat{\epsilon}_i$-DP. Suppose for 10 cameras, $\sum_{i=1}^{N} \hat{\epsilon}_i$ is large enough for the adversary to detect their appearance with high confidence. Then while the adversary can infer that a person appeared *somewhere* across the 10 cameras, the adversary cannot learn *which* cameras they appeared in or when; appearances within individual cameras are still protected by ϵ-DP.

5 PRIVID

In this section, we present PRIVID, a privacy-preserving video analytics system that satisfies (ρ, K, ϵ)-privacy (§2.2 Goals 1 and 2) and provides an expressive query interface which allows analysts to supply their own (untrusted by PRIVID) video-processing code (Goal 3).

5.1 Overview

PRIVID supports *aggregation* queries, which process a "large" amount of video data (e.g., several hours/days of video) and produce a "small" number of bits of output (e.g., a few 32-bit integers). Examples of such tasks include counting the total number of individuals that passed by a camera in one day, or computing the average speed of cars observed. In contrast, PRIVID does not support a query such as reporting the location (e.g., bounding box) of an individual or car within the video frame. PRIVID can be used for one-off ad-hoc queries or standing queries running over a long period, e.g., the total number of cars per day, each day over a year.

The VO decides the level of privacy provided by PRIVID. The VO chooses a privacy policy (ρ, K) and privacy budget (ϵ) for each camera they manage. Given these parameters, PRIVID provides a guarantee of (ρ, K, ϵ)-privacy (Theorem 5.2) for all queries over all cameras it manages.

To satisfy the privacy guarantee, PRIVID utilizes the standard Laplace mechanism from DP [37] to add random noise to the aggregate query result before returning the result to the analyst. The key technical pieces of PRIVID are: (i) providing analysts the ability to specify queries using arbitrary untrusted code (§5.2), (ii) adding noise to results to guarantee (ρ, K, ϵ)-privacy for a single query (§5.5), and (iii) extending the guarantee to handle multiple queries over the same cameras (§5.6).

5.2 PRIVID Query Interface

Execution model. PRIVID requires queries to be expressed using a *split-process-aggregate* model in order to tie the duration of an event to the amount it can impact the query output. The target video is split temporally into chunks, then each chunk is fed to a separate instance of the analyst's processing code, which outputs a set of rows. Together, these rows form a traditional tabular database (untrusted by PRIVID since it is generated by the analyst). The aggregation stage runs a SQL query over this table to produce a raw result. Finally, PRIVID adds noise (§5.5) and returns *only* the noisy result to the analyst, not the raw result or the intermediate table.

Query contents. A PRIVID query must contain (1) a block of statements in a SQL-like language, which we introduce below and call PRIVIDQL, and (2) video processing executables.

(1) PRIVIDQL statements. A valid query contains one or more of *each* of the 3 following statements. We provide an example in §5.7.1 and the full grammar in §E of [33].

● SPLIT statements choose a segment of video (camera, start and end datetime) as input, and produce a set of video chunks as output. They specify how the segment should be split into chunks, i.e., the chunk duration and stride between chunks.

● PROCESS statements take a set of SPLIT chunks as input, and produce a traditional ("intermediate") table. They specify the executable that should process the chunks, the schema of the resulting table, and the maximum number of rows a chunk can output (max_rows, necessary to bound the sensitivity, §5.5). Any rows output beyond the max are dropped.

● SELECT statements resemble typical SQL SELECT statements that operate over the tables resulting from PROCESS statements and output a (ρ,K,ϵ)-private result. They must have an aggregation as the final operation. PRIVID supports the standard aggregation functions (e.g., COUNT, SUM, AVG) and the core set of typical operators as internal relations. An aggregation must specify the range of each column it aggregates (just as in related work on DP for SQL [50]). Each SELECT constitutes at least one data release: one for a single aggregation or multiple for a GROUPBY (one for each key). Each data release receives its own sample of noise and consumes additional privacy budget (§5.6). In order to aggregate across multiple video sources (separate time windows and/or multiple cameras), the query can use a SPLIT and PROCESS for each video source, and then aggregate using a JOIN and GROUPBY in the SELECT.

(2) PROCESS executables. Executables take one chunk as input, and produce a set of rows (e.g., one per object) as output.

5.3 Providing Privacy Despite Blackbox Executables

When running a PRIVID query, an analyst can observe only two pieces of information: (1) the query result, and (2) the time it takes to receive the result.

Query result. In order to link an event's duration to its impact on the output, PRIVID ensures that the output of processing a chunk i can *only* be influenced by what is visible in chunk i (not any other chunk j). Then, an individual can *only* impact the outputs of chunks in which they appear, and the duration of their appearance is directly proportional to their contribution to the output table.

To achieve this, PRIVID processes each chunk using a separate instance of the analyst's executable, each running in its own isolated environment. This environment enforces that the executable can read *only* the video chunk, camera metadata, and a random number generator, and can output *only* values formatted according to the PROCESS schema. However, the executable may use arbitrary operations (e.g., custom ML models for CV tasks).

Execution time. To prevent the execution time from leaking any information, we must add two additional constraints. First, each chunk must complete and return a value within a pre-determined time limit T, otherwise a default value is returned for that chunk (both T and the default value are provided by the analyst at query time).[6] Second, PRIVID only returns the final aggregated query result after $|chunks| \cdot T$. By enforcing these constraints, the observed return time is only a property of the query itself, not the data.

Implementation. Our prototype implementation (described in §D of [33]) satisfies these requirements using standard Linux tools. Alternatively, a deployment of PRIVID could use related work [8, 24, 35] on strong isolation with low overhead.

5.4 Interface Limitations

The main limitation of PRIVID's query interface is the inability to write queries that maintain state across separate chunks. However, in most cases this does not preclude queries, it simply requires them to be expressed in a particular way. One broad class of such queries are those that operate over *unique* objects. Consider a query that counts cars. A straightforward implementation might detect car objects, output one row for each object, and count the number of rows. However, if a car enters the camera view in chunk i and is last visible in chunk $i+n$, the PROCESS table will include n rows for the same car instead of the expected 1. To minimize overcounting, the executable can incorporate a license plate reader, output a plate attribute for each car, and then count(DISTINCT plate) in the SELECT (as in §5.7.1).

Suppose instead the query were counting people, who do not have globally unique identifiers. To minimize overcounting, the PROCESS executable could choose to output a row only for people that *enter* the scene *during that chunk* (and ignore any people that are already visible at the start of a chunk).

PRIVID's aggregation interface imposes some limitations beyond traditional SQL (detailed in §E of [33], e.g., the SELECT must specify the range of each column), but these are equivalent to the limitations of DP SQL interfaces in prior work.

5.5 Query Sensitivity

The sensitivity of a PRIVID query is the maximum amount the final query output could differ given the presence or absence of any (ρ,K)-bounded event in the video. This can be broken down into two questions: (1) what is the maximum number of rows a (ρ, K)-bounded event could impact in the analyst-generated intermediate table, and (2) how much could each of these rows contribute to the aggregate output. We discuss each in turn.

Contribution of a (ρ,K) event to the table. An event that is visible in even a single frame of a chunk can impact the output of that chunk arbitrarily, but due to PRIVID's isolated execution environment, it can *only* impact the output of that chunk, not any others. Thus, the number of rows a (ρ, K)-bounded event could impact is dependent on the number of chunks it spans (an event spans a set of chunks if it is visible in at least one frame of each).

---

[6]Timeouts can impact query accuracy, hence analysts should first profile their code to select a conservative limit T.

In the worst case, an event spans the most contiguous chunks when it is first visible in the last frame of a chunk. Given a chunk duration c (same units as ρ) a single event segment of duration ρ can span at most $\mathsf{max\_chunks}(\rho)$ chunks:

$$\mathsf{max\_chunks}(\rho) = 1 + \lceil \frac{\rho}{c} \rceil \qquad (5.1)$$

DEFINITION 5.1 (Intermediate Table Sensitivity)**.** Consider a privacy policy (ρ, K), and an intermediate table t (created with a chunk size of c_t and maximum per-chunk rows $\mathsf{max\_rows}_t$). The *sensitivity* of t w.r.t (ρ, K), denoted $\Delta_{(\rho, K)}$, is the maximum number of rows that could differ given the presence or absence of any (ρ, K)-bounded event:

$$\Delta_{(\rho, K)}(t) \leq \mathsf{max\_rows}_t \cdot K \cdot \mathsf{max\_chunks}(\rho) \qquad (5.2)$$

Proof. In the worst case, none of the K segments overlap, and each starts at the last frame of a chunk. Thus, each spans a separate $\mathsf{max\_chunks}(\rho)$ chunks (Eq. 5.1). For each of these chunks, all of the $\mathsf{max\_rows}$ output rows could be impacted. □

Sensitivity propagation for (ρ, K)-bounded events. Prior work [45, 50, 57] has shown how to compute the sensitivity of a SQL query over *traditional* tables. Assuming that queries are expressed in relational algebra, they define the sensitivity recursively on the abstract syntax tree. Beginning with the maximum number of rows an individual could influence in the input table, they provide rules for how the influence of an individual propagates through each relational operator and ultimately impacts the aggregation function.

Unlike prior work on propagating sensitivity recursively, the intermediate tables in PRIVID are untrusted, and thus require careful consideration to ensure the privacy definition is rigorously guaranteed. In this work, we determined the set of operations that can be enabled over PRIVID's intermediate tables, derived the sensitivity for each, and proved their correctness. Many rules end up being analogous or similar to those in prior work, but JOINs are different. We provide a brief intuition for these differences below. Fig. 9 in §B contains the complete definition for sensitivity of a PRIVID query.

Privacy semantics of untrusted tables. As an example, consider a query that computes the size of the intersection between two cameras, PROCESS'd into intermediate tables t_1 and t_2 respectively. If $\Delta(t_1) = x$ and $\Delta(t_2) = y$, it is tempting to assume $\Delta(t_1 \cap t_2) = \min(x, y)$, because a value needs to appear in both t_1 and t_2 to appear in the intersection. However, because the analyst's executable can populate the table arbitrarily, they can "prime" t_1 with values that would only appear in t_2, and vice versa. As a result, a value need only appear in either t_1 or t_2 to show up in the intersection, and thus $\Delta(t_1 \cap t_2) = x + y$.

Theorem 5.1. PRIVID's sensitivity definition (Fig. 9, §B) provides (ρ, K, ϵ)-privacy for a query Q over V.
We provide the formal proof in §B.

5.6 Handling Multiple Queries

In traditional DP, the parameter ϵ is viewed as a "privacy budget". Informally, ϵ defines the total amount of information that may be released about a database, and each query consumes a portion of this budget. Once the budget is depleted, no further queries can be answered.

Rather than assigning a single global budget to an entire video, PRIVID allocates a separate budget of ϵ to each frame of a video. When PRIVID receives a query Q over frames $[a, b]$ requesting budget ϵ_Q, it only accepts the query if *all* frames in the interval $[a - \rho, b + \rho]$ have sufficient budget $\geq \epsilon_Q$, otherwise the query is denied (Alg. 1 Lines 1-3). If the query is accepted, PRIVID then subtracts ϵ_Q from each frame in $[a, b]$, but *not* the ρ margin (Alg. 1 Lines 4-5). We require sufficient budget at the ρ margin to ensure that any single segment of an event (which has duration at most ρ) cannot span two temporally disjoint queries (§B).

Note that since each SELECT in a query represents a separate data release, the total budget ϵ_Q used by a query is the sum of the ϵ_i used by each of the i SELECTs. The analyst can specify the amount of budget they would like to use for each release (via a CONSUMING clause, defined in §E of [33], see example in §5.7.1).

Putting it all together. Algorithm 1 presents a simplified (single video) version of the PRIVID query execution process. We provide the full algorithm in §G of [33].

Algorithm 1: PRIVID Query Execution (simplified)

Input : Query Q, video V, interval $[a, b]$, policy (ρ, K, ϵ)
Output: Query answer A

1 **foreach** *frame* $f \in V[a - \rho : b + \rho]$ **do**
2   **if** $f.budget < \epsilon_Q$ **then**
3   **return** DENY

4 **foreach** *frame* $f \in V[a : b]$ **do**
5      $f.budget$ -= $Q.budget$

6 chunks $\leftarrow$ Split $V[a:b]$ into chunks of duration c
7 $T \leftarrow$ Table(schema)
8 **foreach** $chunk \in chunks$ **do**
9   rows $\leftarrow F(chunk)$ // in isolated environment
10   T.append(rows)

11 $r \leftarrow$ execute PrividQL query S over table T
12 $\Delta_{(\rho, K)} \leftarrow$ compute recursively over the structure of S (§5.5)
13 $\eta \leftarrow Laplace(\mu = 0, b = \frac{\Delta}{\epsilon_Q})$
14 $A \leftarrow r + \eta$

Theorem 5.2. Consider an adaptive sequence (§2.3) of n queries $Q_1, ..., Q_n$, each over the same camera C, a privacy policy (ρ_C, K_C), and global budget ϵ_C. PRIVID (Algorithm 1) provides $(\rho_C, K_C, \epsilon_C)$-privacy for all $Q_1, ..., Q_n$.

We provide the formal proof in §B.

5.7 Example Queries

5.7.1 Benevolent Query

Suppose a VO provides access to `camA` via PRIVID, with a policy $(\rho = 60s, K = 2)$. The city transportation department wishes to collect statistics about vehicles passing `camA` during October 2021. We formulate two questions as a PRIVID query:

```
-- Select 1 month time window from camera, split into chunks
SPLIT camA
    BEGIN 10-01-2021/12:00am END 11-01-2021/12:00am
    BY TIME 10sec STRIDE 0sec
    INTO chunksA;
-- Process chunks using analyst's code, store outputs in tableA
PROCESS chunksA USING traffic_flow.py TIMEOUT 1sec
    PRODUCING 20 ROWS
    WITH SCHEMA (plate:STRING="", type:STRING="", speed:NUMBER=0)
    INTO vehiclesA;
-- S1: Number of unique cars per day
SELECT day,COUNT(DISTINCT plate) FROM vehiclesA WHERE type=="car"
        GROUP BY day CONSUMING eps=0.5;
-- S2: Average speed of trucks
SELECT AVG(range(speed, 30, 60)) FROM vehiclesA WHERE type=="truck"
        CONSUMING eps=0.5;
```

The SPLIT selects 1 month of video from `camA`, then divides the frames into a list of 10-second-long chunks (267k chunks total). The PROCESS first creates an empty table based on the SCHEMA (3 columns). Then, for each chunk, it starts a fresh instance of `traffic_flow.py` inside a restricted container, provides the chunk as input, and appends the output as rows to `vehiclesA`. The executable `traffic_flow.py` contains off-the-shelf object detection and tracking models, a license plate reader, and a speed estimation algorithm (source in §F of [33]).

The first SELECT filters all cars, then counts the "distinct" license plates to estimate the number of *unique* cars per day. Each day is a separate data release with an independent sample of noise. The second SELECT filters all trucks, then computes the average speed across the entire month of footage. It uses the same input video as the first select, and thus draws from the same budget, so in aggregate the two SELECTs consume $\epsilon = 1.0$ budget from all frames in October 2021.

5.7.2 Malicious Query Attempt

Now consider a malicious analyst Mal who wishes to determine if individual x appeared in front of `camA` each day. Assume x's appearance is bound by the VO's (ρ, K) policy.

To hide their intent, Mal disguises their query as a traffic counter, mimicking S_1 from the previous example. They write identical query statements, but their "`traffic_flow.py`" instead includes specialized models to detect x. If x appears, it outputs 20 rows (the maximum) with random values for each of the columns, otherwise it outputs 0 rows. This adds 20 rows to the corresponding daily count for each chunk x appears.

Amplification attempt. Due to the isolated environment (§5.3), the PROCESS executable can only output rows for a chunk if x *truly appears*. It has no way of saving state or communicating between executions in order to artificially output rows for a chunk in which x does not appear. It could output more than 20 rows for a single chunk, but PRIVID ignores any rows beyond the PROCESS's explicit max (20), so this would not

increase the count. Increasing the rows per chunk parameter would also be pointless: PRIVID would compute a proportionally higher sensitivity and add proportionally higher noise.

Side channel attempt. The executable could try to encode the entire contents of a frame in a row of the table, either by encoding it as a string, or a very large number of individual integer columns. But in either case, the analyst cannot view the table directly or even a single row directly, it can only compute noisy aggregations over entire columns.

Summary. PRIVID would compute the sensitivity of S_1 (identical in both the benevolent and malicious cases) as $\Delta_{(60,2)}(Q) \leq 20 \cdot 2 \cdot (1 + \lceil \frac{60}{10} \rceil) = 280$ rows, meaning it would add noise with scale 280 to each daily count. Regardless of how Mal changes her executable, it cannot output more than 280 rows based on x's presence. Thus, even if she observed a non-zero value ~ 280, she could not distinguish whether it is a result of the noise or x's appearance.

Mal's query gets a useless result, because her target (x's appearance) was close in duration to the policy. In contrast, the benevolent query can get a useful result because the duration of its target (the set of *all cars'* appearances) far exceeds the policy. PRIVID's noise will translate to $\mathcal{L}^{-1}(p = 0.99, u = 0, b = \frac{\Delta}{\epsilon} = \frac{280}{0.5}) \leq 2200$ cars with 99% confidence. If, for example, there are an average of 10 cars in each chunk (and thus 86000 in one day), 2200 represents an error of $\pm 2.5\%$.

6 Query Utility Optimization

The noise that PRIVID adds to a query result is proportional to both the privacy policy (ρ, K) and the range of the aggregated values (the larger the range, the more noise PRIVID must add to compensate for it). In this section we introduce two optional optimizations that PRIVID offers analysts to improve query accuracy while maintaining an equivalent level of privacy: one reduces the ρ needed to preserve privacy (§6.1), while the other reduces the range for aggregation (§6.2).

6.1 Spatial Masking

Observation. In certain settings, a few individuals may be visible to a camera for far longer than others (e.g., those sitting on a bench or in a car), creating a heavy-tailed distribution of presence durations. Fig. 3 (top row) provides some representative examples. Setting (ρ, K) to the maximum

| (a) campus | (b) highway | (c) urban |

Figure 3: (Top) Heatmap measuring the maximum time any object spent in each pixel, noramlized to the max (yellow) per video. (Bottom) The resulting masks used for our evaluation, chosen from the list of masks automatically generated using the algorithm in §I of [33].
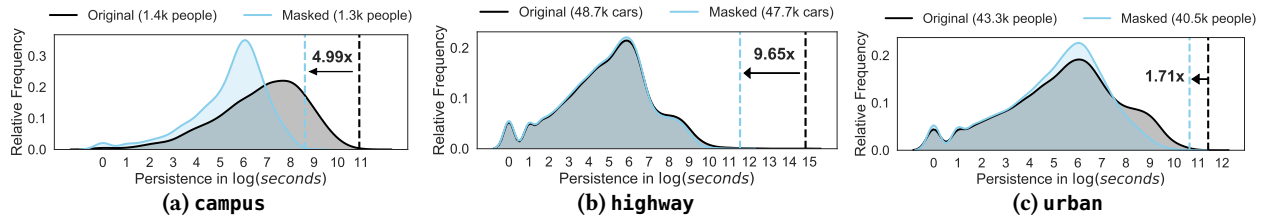
Figure 4: The distribution of private objects' durations (persistence) is heavy tailed. Applying the mask from Fig. 3 significantly lowers the maximum duration, while still allowing most private objects to be detected. The key denotes the total number of private objects detectable before and after applying the mask. The dotted lines highlight the maximum persistence, and the arrow text denotes the relative reduction.

duration in such distributions would result in a large amount of noise needed to protect just those few individuals; all others could have been protected with a far lower amount of noise. We observe that, in many cases, lingering individuals tend to spend the majority of their time in one of a few fixed regions in the scene, but a relatively short time in the rest of the scene. For example, a car may be parked in a spot for hours, but only visible for 1 minute while entering/leaving the spot.

Opportunity. Masking fixed regions (i.e., removing those pixels from all frames prior to running the analyst's video processing) in the scene that contain lingering individuals would drastically reduce the *observable* maximum duration of individuals' presence, e.g., the parked car from above would be observable for 1 minute rather than hours. This, in turn, would permit a policy with a smaller ρ, but an equivalent level of privacy–all appearances would still be bound by the policy. Of course, this technique is only useful to an analyst when the remaining (unmasked) part of the scene includes all the information needed for the query at hand, e.g., if counting cars, masking sidewalks would be reasonable but masking roads would not.

Optimization. At camera-registration time, instead of providing a single (ρ, K) policy per camera, the VO can provide a (fixed) list of a few frame masks and, for each, a corresponding (ρ, K) policy that would provide equivalent privacy when that mask is applied. At query time, the analyst can (optionally) choose a mask from the list that would minimally impact their query goal while maximizing the level of noise reduction (via the tighter (ρ, K) bound). If a mask is chosen, PRIVID applies it to all video frames *before* passing it to the analyst's PROCESS executable (the analyst only "sees" the masked video), and uses the corresponding (ρ, K) in the sensitivity calculation (§5.5).

To aid the analyst in discovering a useful set of masks (i.e., those that reduce (ρ, K) as much as possible using the fewest pixels), we provide an algorithm in §I.2 of [33]. Regardless of how they are chosen, the masks themselves are static (i.e., the same pixels are masked in every frame regardless of its contents), and the set of available masks is fixed. Neither depend on the query or the target video. Further, the mask itself does not reveal how the analyst generated it or which specific objects contributed to it, it only tells the analyst that some objects appear for a long duration in the masked region.

Noise reduction. We demonstrate the potential benefit of masking on three queries (Q1-Q3) from our evaluation

Video	Max(frame)	Max(region)	Reduction
campus	6	3	2.00×
highway	40	23	1.74×
urban	37	16	2.25×

Table 2: Reduction in max output range from splitting each video into distinct regions. Reduction shows the factor by which the noise could be reduced. 2× cuts the necessary privacy level in half.

(Table 3). Given the query tasks (counting unique people and cars), we chose masks that would maximally reduce ρ without impacting the object counts; the bottom row of Fig. 3 visualizes our masks. Fig. 4 shows that these masks reduce maximum durations by 1.71-9.65×. In §I.1 of [33] we show that masking provides similar benefits for 7 additional videos evaluated by BlazeIt [47] and MIRIS [30].

Masking vs. denaturing. Although masking is a form of denaturing, PRIVID uses it differently than the prior approaches in §3.1, in order to sidestep their issues. Rather than attempting to dynamically hide individuals as they move through the scene, PRIVID's masks cover a *fixed* location in the scene and are publicly available so analysts can account for them in their query implementation. Also, masks are used as an optional modification to the input video; the rest of the PRIVID pipeline, and thus its formal privacy guarantees, remain the same.

6.2 Spatial Splitting

Observation. (1) At any point in time, each object typically occupies a relatively small area of a video frame. (2) Many common queries (e.g., object detections) do not need to examine the entire contents of a frame at once, i.e., if the video is split spatially into regions, they can compute the same total result by processing each of the regions separately.

Opportunity. PRIVID already splits videos temporally into chunks. If each chunk is further divided into spatial regions and an individual can only appear in one of these chunks at a time, then their presence occupies a relatively smaller portion of the intermediate table (and thus requires less noise to protect). Additionally, the maximum duration of each individual region may be smaller than the frame as a whole.

Optimization. At camera-registration time, PRIVID allows VOs to manually specify boundaries for dividing the scene into regions. They must also specify whether the boundaries are soft (individuals may cross them over time, e.g., between two crosswalks) or hard (individuals will never cross them, e.g., between opposite directions on a highway). At query

time, analysts can optionally choose to spatially split the video using these boundaries. Note that this is in addition to, rather than in replacement of, the temporal splitting. If the boundaries are soft, tables created using that split must use a chunk size of 1 frame to ensure that an individual can always be in at most 1 chunk. If the boundaries are hard, there are no restrictions on chunk size since the VO has stated the constraint will always be true.

Noise reduction. We demonstrate the potential benefit of spatial splitting on three videos from our evaluation (Q1-Q3). For each video, we manually chose intuitive regions: a separate region for each crosswalk in `campus` and `urban` (2 and 4, respectively), and a separate region for each direction of the road in `highway`. Table 2 compares the range necessary to capture all objects that appear within one chunk in the entire frame compared to the individual regions. The difference (1.74-2.25×) represents the potential noise reductions from splitting: noise is proportional to $\mathrm{max(frame)}$ or $\mathrm{max(region)}$ when splitting is disabled or enabled, respectively.

Grid split. To increase the applicability of spatial splitting, PRIVID could allow analysts to divide each frame into a grid and remove the restrictions on soft boundaries to allow any chunk size. This would require additional estimates about the max size of any private object (dictating the max number of regions they could occupy at any time), and the maximum speed of any object across the frame (dictating the max number of regions they could move between). We leave this to future work.

7 Evaluation

The evaluation highlights of PRIVID are as follows:

1. PRIVID supports a diverse range of video analytics queries, including object counting, duration queries, and composite queries; for each, PRIVID increases error by 1-21% relative to a non-private system, while protecting all individuals with (ρ, K, ϵ)-privacy (§7.2).
2. PRIVID enables VOs and analysts to flexibly and formally trade utility loss and query granularity while preserving the same privacy guarantee (§7.3).

7.1 Evaluation Setup

Datasets. We evaluated PRIVID primarily using three representative video streams (`campus`, `highway` and `urban`, screenshots in Fig. 3) that we collected from YouTube spanning 12 hours each (6am-6pm). For one case study (multi-camera), we use the Porto Taxi dataset [58] containing 1.7mil trajectories of all 442 taxis running in the city of Porto, Portugal from Jan. 2013 to July 2014. We apply the same processing as [42] to emulate a city-wide camera dataset; the result is the set of timestamps each taxi would have been visible to each of 105 cameras over the 1.5 year period.

Implementation. We implemented PRIVID in 4k lines of Python. We used the Faster-RCNN [63] model in Detectron-v2 [71] for object detection, and DeepSORT [69] for object tracking. For these models to work reasonably given the di-

verse content of the videos, we chose hyperparameters for detection and tracking on a per-video basis (details in §H of [33]).

Privacy policies. We assume the VO's underlying privacy goal is to "protect the appearance of all individuals". For each camera, we use the strategy in §6.1, to create a map between masks and (ρ, K) policies that achieve this goal.

Query parameters. For each query, we first chose a mask that covered as much area as possible (to get the minimal ρ) without disrupting the query. The resulting ρ values are in Table 3. We use a budget of $\epsilon = 1$ for each query. We chose query windows sizes (W), chunk durations (c), and column ranges to best approximate the analyst's expectations for each query (as opposed to picking optimal values based on a parameter sweep, which the analyst is unable to do).

Baselines. For each query, we compute `error` by comparing the output of PRIVID to running the same exact query implementation without PRIVID. We execute each query 1000 times, and report the mean accuracy value $\pm$ 1 standard deviation.

7.2 Query Case Studies

We formulate five types of queries to span a variety of axes (target object class, number of cameras, aggregation type, query duration, standing vs. one-off query). Fig. 5 displays results for Q1-Q3. Table 3 summarizes the remaining queries (Q4-Q13).

Case 1: Q1-Q3 (Counting private objects over time). To demonstrate PRIVID's support for standing queries and short (1 hour) aggregation durations, we `SUM` the number of *unique* objects observed *each hour* over the 12 hours.

Case 2: Q4-Q6 (Aggregating over multiple cameras with complex operators). We utilize `UNION`, `JOIN`, and `ARGMAX` to aggregate over cameras in the `Porto Taxi Dataset`. Due to the large aggregation window (1 year), PRIVID's noise addition is small (relative to the other queries using a window on the order of hours) and accuracy is high.

Case 3: Q7-Q9 (Counting non-private objects, large window). We measure the fraction of trees (non-private objects) that have bloomed in each video. Executed over an entire network of cameras, such a query could be used to identify the regions with the best foliage in Spring. Relative to Case 1, we achieve high accuracy by using a longer query window of 12 hours (the status of a tree does not change on that time scale), and minimal chunk size (1 frame, no temporal context needed).

Case 4: Q10-Q12 (Fine-grained results using aggressive masking). We measure the average amount of time a traffic signal stays red. Since this only requires observing the light itself, we can mask *everything else*, resulting in a ρ bound of 0 (no private objects overlap these pixels), enabling high accuracy and fine temporal granularity.

Case 5: Q13 (Stateful query). We count only the individuals that enter from the south and exit at the north. It requires a larger chunk size (relative to Q1-Q3) to maintain enough state within a single chunk to understand trajectory.

Case #	Q#	Query Description	Query Parameters	Video	ρ	Query Output	Error
Case 2	Q4	Average Taxi Driver Working Hours (`union` across 2 cameras)	$\|W\|=365$ days, $c=15$ sec, Agg = `avg`, range $=(0,16)$	`porto10, porto27`	[45, 195] sec	5.87 hrs	5.86%±0.18%
Case 2	Q5	Average # Taxis Traversing 2 Locations on Same Day (`intersection` across 2 cameras)	$\|W\|=365$ days, $c=15$ sec, Agg = `avg`, range $=(0,300)$	`porto10, porto27`	[45, 195] sec	131 taxis	0.20%±0.13%
Case 2	Q6	Identifying Camera with Highest Daily Traffic (`argmax` across all 105 cameras)	$\|W\|=365$ days, $c=15$ sec, Agg = `argmax`	`porto0, ..., porto104`	[15, 525] sec	`porto20`	0%
Case 3	Q7	Fraction of trees with leaves (%)	$\|W\|=12$ hrs, $c=1$ frame, Agg = `avg`, range $=(0,100)$	campus	49 sec	15/15 = 1.00	0.10%±0.11%
Case 3	Q8			highway	6.21 min	3/7 = 0.43	1.76%±1.90%
Case 3	Q9			urban	3.34 min	4/6 = 0.67	0.61%±0.66%
Case 4	Q10	Duration of Red Light (seconds)	$\|W\|=12$ hrs, $c=30$ min, Agg = `avg`, range $=(0,300)$	campus	1 frame	75 sec	0%±1.4×10⁻⁴%
Case 4	Q11			highway	1 frame	50 sec	0%±2.1×10⁻⁴%
Case 4	Q12			urban	1 frame	100 sec	0%±1.0×10⁻⁴%
Case 5	Q13	# Unique People (Filter: trajectory moving towards campus)	$\|W\|=12$ hrs, $c=10$ sec, Agg = `sum`, range $=(0,5)$	campus	49 sec	576 people	20.31%±2.60%

Table 3: Summary of query results for Q4-Q13. For Case 3 and 5, we use the same masks (and thus ρ) from Fig. 3. For Case 4, we mask all pixels except the traffic light to attain $\rho = 0$. For Case 2 we do not use any masks.
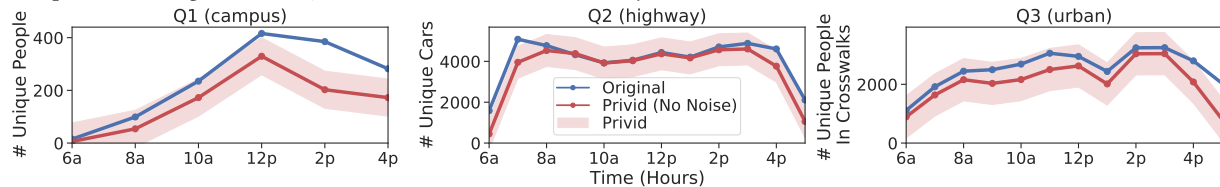
Figure 5: Time series of PRIVID's output for Case 1 queries. "Original" is the baseline query output without using PRIVID. "Privid (No Noise)" shows the raw output of PRIVID before noise is added. The final noisy output will fall within the range of the red ribbon 99% of the time.
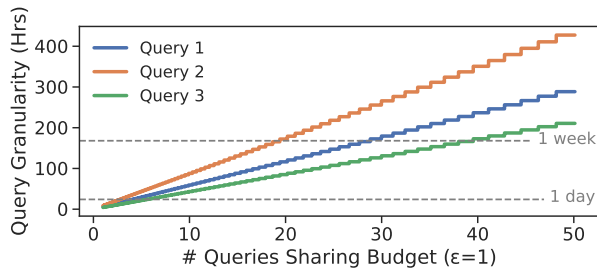
Figure 6: Given a fixed query and accuracy target, decreasing the amount of budget used by each query allows more queries to be executed over the same video segment, but requires a proportionally coarser granularity. The x-axis plots the number of queries evenly sharing a budget of $\epsilon = 1$, thus $x = 10$ means 10 instances of the same exact query over the same video segment, each using a budget of $\frac{1}{10}$. We fix the accuracy target to be 99% of values having error $\leq 5\%$.
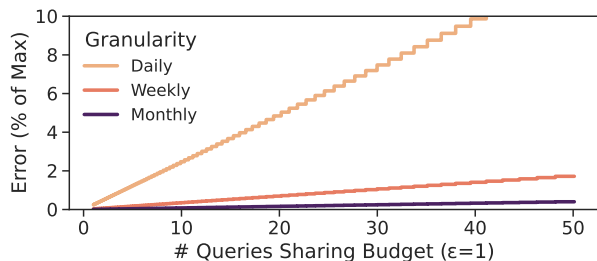
Figure 7: Given a fixed query and granularity, decreasing the amount of budget used by each query allows more queries to be executed over the same video segment, but results in proportionally higher error. The x-axis is the same as Fig. 6. Each line corresponds to Q1 using a different granularity. The y-axis plots the error for 99% of values. Error is the amount of noise added relative to the maximum query output. For example, in Q4, the final output is the average number of working hours in the range [0,16]. Thus an error of 1% would mean the noisy result is within 0.16 hours of the true result.

7.3 Budget-Granularity Tradeoff

Analysts have two main knobs for each query Q to navigate the utility space: (1) the fraction ϵ_Q of the total budget ϵ used by that query, and (2) the duration (granularity) of each aggregation (i.e., "one value per day for a month" has a granularity of one day). The query budget is inversely proportional to both the query granularity and error (the expected value of noise PRIVID adds relative to the output range). Thus, to decrease the amount of budget per query (or equivalently, increase the number of queries sharing the budget), an analyst must choose a (temporally) coarser result, a larger expected error bound, or both. Fig. 6 shows that, for example, 5 instances of Query 3 could release results daily or 40 instances of Query 3 could release results weekly, while achieving the same expected accuracy. Fig. 7 shows that, for example, 20 separate instances of Query 1 ($x = 20$) executed over the same target video could each expect 4.8% error if they release one result daily, 0.7% error if they release one weekly, or 0.16% error if they release one monthly. Importantly, this tradeoff is transparent to analysts: Figs. 6 and 7 rely only on information that is publicly available to analysts and did not require executing any queries.

7.4 Analyzing Sources of Inaccuracy

PRIVID introduces two sources of inaccuracy to a query result: (1) intentional noise to satisfy (ρ, K, ϵ)-privacy, and (2) (unintentional) inaccuracies caused by the impact of splitting and masking videos before executing the video processing. Fig. 5 shows these two sources separately for queries Q1-Q3 (Case 1): the discrepancy between the two curves demonstrates the impact of (2), while the shaded belt shows the relative scale of noise added (1). In summary, the scale of noise added by PRIVID allows the final result to preserve the trend of the original.

8 Using PRIVID

In this section, we summarize the set of decisions that both the VO and analyst need to make when interacting with PRIVID.

8.1 Video Owner

First, the VO must register a set of cameras with PRIVID. For *each* camera, they must supply: (1) a (ρ, K) bound (or more generally a map of masks to bounds), (2) a privacy budget ϵ, and (3) some metadata describing the scene to analysts (e.g., a short video clip, since they cannot view the camera feed directly). All of this is public to analysts. Below we provide general suggestions for the VO, but ultimately they are responsible for choosing these values. PRIVID only handles enforcing a given policy.

(1) (ρ, K) bounds. In most cases, we expect the VO will record a sample of video, measure durations of objects of interest using off-the-shelf tracking algorithms, and then set the bound to the longest duration.

To provide better utility for analysts, the VO can offer a menu of static masks that remove some of the scene in exchange for tighter noise bounds than the original policy (which is itself mapped to the empty mask). Note that the VO must explicitly choose a (ρ, K) policy for each mask. A mask is only useful if it reduces the amount of time the longest objects are visible, which enables a tighter bound while protecting the same set of individuals.

The VO may draw masks manually or generate them automatically, e.g., by analyzing past trends from the camera. In general, we expect masks to be static properties of each scene, dependent only on dynamics of the scene type, rather than behaviors of any individuals. However, it is ultimately the VO's responsibility to ensure any masks it provides do not reveal anything private, such as a person's silhouette. PRIVID focuses on preventing the leakage of privacy when answering queries. It does not make any guarantees about the mask itself.

(2) Budget ϵ. As in any deployment of DP, the choice of ϵ is subjective. Academic papers commonly use $\epsilon \approx 1$ [52] while recent industry deployments have used $1 < \epsilon < 10$ [27, 36, 56]. Note that in PRIVID, this budget is *per-frame* (§5.6); two queries aggregating over disjoint time ranges of the same video draw from separate budgets. The only PRIVID-specific consideration for choosing ϵ is that cameras with overlapping fields of view should share the same budget.

(3) Metadata. The VO should release a sample video clip[7] representative of the scene so that analysts can calibrate their executable[8] and query[9] accordingly. Any privacy loss resulting from the one-time release of this single clip is limited, and can be manually vetted by the VO. Optionally, the VO can release additional information to aid analysts, such as the camera's GPS coordinates, make, or focal length settings.

---

[7]While a clip is not needed in principle, without it, the analyst "runs blind" and will not have confidence in the correctness of their results.

[8]ML models may perform better when retrained on a particular scene.

[9]For example, queries must specify bounds on the amount of output per chunk, which depend on the amount of activity in the scene.

8.2 Analyst

In order to formulate a PRIVID query the analyst must make the following decisions. For each decision, we provide an example for the query in §5.7.1 (counting cars crossing a virtual line on a highway).

Choose a mask (from the list provided by the VO) based on the query goal. For example, they should select a mask that covers as much of the scene as possible without covering the area near the virtual line. This would significantly reduce the bound by removing parking spots and intersections where objects linger.

Choose a chunk size based on the amount of context needed. A larger chunk size permits more context for each execution of the PROCESS, but results in more noise (§5.5). Thus, the analyst should choose the smallest chunk size that captures their events of interest. For example, 1 second is likely sufficient to capture cars driving past a line. If they instead wanted to calculate car speed, they would need a larger chunk size (e.g., 10 seconds) and less restrictive mask to capture more of the car's trajectory.

Choose upper bound on number of output rows per chunk based on the expected (via the video sample) level of activity in each chunk. For counting cars over a short chunk, especially in less busy scenes, each chunk may see 1-2 cars and thus need 1-2 rows. For calculating speed over a larger chunk, especially in more busy scenes, each chunk will see more cars and may need 10 or 100 rows.

Create a PROCESS executable. This involves tuning their CV models based on the scene (via the sample video), and combining all tasks into a single executable. For example, their executable may include an object detector to find cars, an object tracker to link them to trajectories, a license plate reader to link cars across cameras or prevent double counting, and an algorithm to compute speed or determine car model.

Choose query granularity and budget. The query granularity and budget are directly proportional to accuracy. Given a fixed value for each, improving one requires worsening another proportionally. We elaborate upon this tradeoff in §7.3.

9 Ethics

In building PRIVID, we *do not* advocate for the increase of public video surveillance and analysis. Instead, we observe that it is already prevalent and seek to improve the privacy landscape. PRIVID's accuracy and expressiveness makes it palatable to add formal privacy to existing analytics, and lowers the barrier to deployment. If privacy legislation is introduced, PRIVID could be one way to ensure compliance.

References

[1] Absolutely everywhere in beijing is now covered by police video surveillance. https://qz.com/518874/.

[2] Are we ready for ai-powered security cameras? https://thenewstack.io/are-we-ready-for-ai-powered-security-cameras/.

[3] British transport police: Cctv. http://www.btp.police.uk/advice_and_information/safety_on_and_near_the_railway/cctv.aspx.

[4] Can 30,000 cameras help solve chicago's crime problem? https://www.nytimes.com/2018/05/26/us/chicago-police-surveillance.html.

[5] Data generated by new surveillance cameras to increase exponentially in the coming years. http://www.securityinfowatch.com/news/12160483/.

[6] Detection leaderboard. https://cocodataset.org/#detection-leaderboard.

[7] Epic domestic surveillance project. https://epic.org/privacy/surveillance/.

[8] nsjail. https://github.com/google/nsjail.

[9] Oakland bans use of facial recognition. https://www.sfchronicle.com/bayarea/article/Oakland-bans-use-of-facial-recognition-14101253.php.

[10] Paris hospitals to get 1,500 cctv cameras to combat violence against staff. https://bit.ly/2OYiBz2.

[11] Powering the edge with ai in an iot world. https://www.forbes.com/sites/forbestechcouncil/2020/04/06/powering-the-edge-with-ai-in-an-iot-world/.

[12] San francisco is first us city to ban facial recognition. https://www.bbc.com/news/technology-48276660.

[13] Video analytics applications in retail - beyond security. https://www.securityinformed.com/insights/co-2603-ga-co-2214-ga-co-1880-ga.16620.html/.

[14] The vision zero initiative. http://www.visionzeroinitiative.com/.

[15] What's wrong with public video surveillance? https://www.aclu.org/other/whats-wrong-public-video-surveillance, 2002.

[16] Abuses of surveillance cameras. http://www.notbored.org/camera-abuses.html, 2010.

[17] Mission creep-y: Google is quietly becoming one of the nation's most powerful political forces while expanding its information-collection empire. https://www.citizen.org/wp-content/uploads/google-political-spending-mission-creepy.pdf, 2014.

[18] Mission creep. https://www.aclu.org/other/whats-wrong-public-video-surveillance, 2017.

[19] How retail stores can streamline operations with video content analytics. https://www.briefcam.com/resources/blog/how-retail-stores-can-streamline-operations-with-video-content-analytics/, 2020.

[20] The mission creep of smart streetlights. https://www.voiceofsandiego.org/topics/public-safety/the-mission-creep-of-smart-streetlights/, 2020.

[21] Video analytics traffic study creates baseline for change. https://www.govtech.com/analytics/Video-Analytics-Traffic-Study-Creates-Baseline-for-Change.html, 2020.

[22] What is computer vision? ai for images and video. https://www.infoworld.com/article/3572553/what-is-computer-vision-ai-for-images-and-video.html, 2020.

[23] P. Aditya, R. Sen, P. Druschel, S. Joon Oh, R. Benenson, M. Fritz, B. Schiele, B. Bhattacharjee, and T. T. Wu. I-pic: A platform for privacy-compliant image capture. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '16, page 235–248, New York, NY, USA, 2016. Association for Computing Machinery.

[24] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th {usenix} symposium on networked systems design and implementation ({nsdi} 20)*, pages 419–434, 2020.

[25] Amazon. Rekognition. https://aws.amazon.com/rekognition/.

[26] G. Ananthanarayanan, Y. Shu, M. Kasap, A. Kewalramani, M. Gada, and V. Bahl. Live video analytics with microsoft rocket for reducing edge compute costs, July 2020.

[27] Apple Differential Privacy Team. Learning with privacy at scale. *Apple Machine Learning Journal*, 1(8), 2017.

[28] M. Azure. Computer vision api. https://azure.microsoft.com/en-us/services/cognitive-services/computer-vision/, 2021.

[29] M. Azure. Face api. https://azure.microsoft.com/en-us/services/cognitive-services/face/, 2021.

[30] F. Bastani, S. He, A. Balasingam, K. Gopalakrishnan, M. Alizadeh, H. Balakrishnan, M. Cafarella, T. Kraska, and S. Madden. Miris: Fast object track queries in video. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1907–1921, New York, NY, USA, 2020. Association for Computing Machinery.

[31] A. Bewley, Z. Ge, L. Ott, F. Ramos, and B. Upcroft. Simple online and realtime tracking. In *2016 IEEE International Conference on Image Processing (ICIP)*, pages 3464–3468, 2016.

[32] Z. Cai, M. Saberian, and N. Vasconcelos. Learning complexity-aware cascades for deep pedestrian detection. In *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*, ICCV '15, pages 3361–3369, Washington, DC, USA, 2015. IEEE Computer Society.

[33] F. Cangialosi, N. Agarwal, V. Arun, J. Jiang, S. Narayana, A. Saarwate, and R. Netravali. Privid: Practical, privacy-preserving video analytics queries (extended version). https://arxiv.org/abs/2106.12083.

[34] A. Chattopadhyay and T. E. Boult. Privacycam: a privacy preserving camera using uclinux on the blackfin dsp. In *2007 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8. IEEE, 2007.

[35] G. Chrome. minijail0. https://google.github.io/minijail/.

[36] B. Ding, J. Kulkarni, and S. Yekhanin. Collecting telemetry data privately. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 3571–3580. Curran Associates, Inc., 2017.

[37] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In S. Halevi and T. Rabin, editors, *Theory of Cryptography*, volume 3876 of *Lecture Notes in Computer Science*, pages 265–284, Berlin, Heidelberg, Mar. 2006. Springer.

[38] I. Ghodgaonkar, S. Chakraborty, V. Banna, S. Allcroft, M. Metwaly, F. Bordwell, K. Kimura, X. Zhao, A. Goel, C. Tung, et al. Analyzing worldwide social distancing through large-scale computer vision. *arXiv preprint arXiv:2008.12363*, 2020.

[39] Google. Cloud vision api. https://cloud.google.com/vision, 2021.

[40] K. Hsieh, G. Ananthanarayanan, P. Bodik, S. Venkataraman, P. Bahl, M. Philipose, P. B. Gibbons, and O. Mutlu. Focus: Querying large video datasets with low latency and low cost. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 269–286, 2018.

[41] IBM. Maximo remote monitoring. https://www.ibm.com/products/maximo/remote-monitoring, 2021.

[42] S. Jain, G. Ananthanarayanan, J. Jiang, Y. Shu, and J. E. Gonzalez. Scaling Video Analytics Systems to Large Camera Deployments. In *ACM HotMobile*, 2019.

[43] S. Jain, X. Zhang, Y. Zhou, G. Ananthanarayanan, J. Jiang, Y. Shu, V. Bahl, and J. Gonzalez. Spatula: Efficient cross-camera video analytics on large camera networks. In *ACM/IEEE Symposium on Edge Computing (SEC 2020)*, November 2020.

[44] J. Jiang, G. Ananthanarayanan, P. Bodik, S. Sen, and I. Stoica. Chameleon: scalable adaptation of video analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 253–266. ACM, 2018.

[45] N. Johnson, J. P. Near, and D. Song. Towards practical differential privacy for sql queries. *Proceedings of the VLDB Endowment*, 11(5):526–539, 2018.

[46] P. Kairouz, S. Oh, and P. Viswanath. The composition theorem for differential privacy. *IEEE Transactions on Information Theory*, 63(6):4037–4049, 2017.

[47] D. Kang, P. Bailis, and M. Zaharia. Blazeit: optimizing declarative aggregation and limit queries for neural network-based video analytics. *Proceedings of the VLDB Endowment*, 13(4):533–546, 2019.

[48] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia. Noscope: optimizing neural network queries over video at scale. *Proceedings of the VLDB Endowment*, 10(11):1586–1597, 2017.

[49] D. Kang, J. Guibas, P. Bailis, T. Hashimoto, and M. Zaharia. Task-agnostic indexes for deep learning-based queries over unstructured data. *arXiv preprint arXiv:2009.04540*, 2020.

[50] I. Kotsogiannis, Y. Tao, X. He, M. Fanaeepour, A. Machanavajjhala, M. Hay, and G. Miklau. Privatesql: A differentially private sql query engine. *Proc. VLDB Endow.*, 12(11):1371–1384, July 2019.

[51] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017.

[52] Y.-H. Kuo, C.-C. Chiu, D. Kifer, M. Hay, and A. Machanavajjhala. Differentially private hierarchical count-of-counts histograms. *Proceedings of the VLDB Endowment*, 11.11:1509—1521, 2018.

[53] H. Li, Z. Lin, X. Shen, J. Brandt, and G. Hua. A convolutional neural network cascade for face detection. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5325–5334, June 2015.

[54] Y. Li, A. Padmanabhan, P. Zhao, Y. Wang, G. H. Xu, and R. Netravali. Reducto: On-Camera Filtering for Resource-Efficient Real-Time Video Analytics. SIGCOMM '20, page 359–376, New York, NY, USA, 2020. Association for Computing Machinery.

[55] T. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie. Feature pyramid networks for object detection. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 936–944, July 2017.

[56] A. Machanavajjhala, D. Kifer, J. M. Abowd, J. Gehrke, and L. Vilhuber. Privacy: Theory meets practice on the map. In *ICDE*, 2008.

[57] F. D. McSherry. Privacy integrated queries: An extensible platform for privacy-preserving data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, page 19–30, New York, NY, USA, 2009. Association for Computing Machinery.

[58] L. Moreira-Matias, J. Gama, M. Ferreira, J. Mendes-Moreira, and L. Damas. Predicting taxi–passenger demand using streaming data. *IEEE Transactions on Intelligent Transportation Systems*, 14(3):1393–1402, 2013.

[59] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of aes. In *Cryptographers' track at the RSA conference*, pages 1–20. Springer, 2006.

[60] J. R. Padilla-López, A. A. Chaaraoui, and F. Flórez-Revuelta. Visual privacy protection methods: A survey. *Expert Systems with Applications*, 42(9):4177–4195, 2015.

[61] C. Percival. Cache missing for fun and profit, 2005.

[62] R. Poddar, G. Ananthanarayanan, S. Setty, S. Volos, and R. A. Popa. Visor: Privacy-preserving video analytics as a cloud service. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 1039–1056, 2020.

[63] S. Ren, K. He, R. B. Girshick, and J. Sun. Faster R-CNN: towards real-time object detection with region proposal networks. *CoRR*, abs/1506.01497, 2015.

[64] J. Stanley and A. C. L. Union. *The Dawn of Robot Surveillance: AI, Video Analytics, and Privacy*. American Civil Liberties Union, 2019.

[65] Y. Sun, X. Wang, and X. Tang. Deep convolutional network cascade for facial point detection. In *Proceedings of the 2013 IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '13, pages 3476–3483, Washington, DC, USA, 2013. IEEE Computer Society.

[66] H. Wang, Y. Hong, Y. Kong, and J. Vaidya. Publishing video data with indistinguishable objects. *Advances in database technology : proceedings. International Conference on Extending Database Technology*, 2020:323 – 334, 2020.

[67] H. Wang, S. Xie, and Y. Hong. Videodp: A universal platform for video analytics with differential privacy. *arXiv preprint arXiv:1909.08729*, 2019.

[68] J. Wang, B. Amos, A. Das, P. Pillai, N. Sadeh, and M. Satyanarayanan. A scalable and privacy-aware iot service for live video analytics. In *Proceedings of the 8th ACM on Multimedia Systems Conference*, pages 38–49. ACM, 2017.

[69] N. Wojke, A. Bewley, and D. Paulus. Simple online and realtime tracking with a deep association metric. In *2017 IEEE International Conference on Image Processing (ICIP)*, pages 3645–3649. IEEE, 2017.

[70] H. Wu, X. Tian, M. Li, Y. Liu, G. Ananthanarayanan, F. Xu, and S. Zhong. Pecam: Privacy-enhanced video streaming and analytics via securely-reversible transformation. In *ACM MobiCom*, October 2021.

[71] Y. Wu, A. Kirillov, F. Massa, W.-Y. Lo, and R. Girshick. Detectron2. https://github.com/facebookresearch/detectron2, 2019.

[72] X. Yu, K. Chinomi, T. Koshimizu, N. Nitta, Y. Ito, and N. Babaguchi. Privacy protecting visual processing for secure video surveillance. In *2008 15th IEEE International Conference on Image Processing*, pages 1672–1675. IEEE, 2008.

[73] H. Zhang, G. Ananthanarayanan, P. Bodik, M. Philipose, P. Bahl, and M. J. Freedman. Live video analytics at scale with approximation and delay-tolerance. In *NSDI*, volume 9, page 1, 2017.

[74] X. Zhu, Y. Wang, J. Dai, L. Yuan, and Y. Wei. Flow-guided feature aggregation for video object detection. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 408–417, 2017.

A Relative Privacy Guarantees

A.1 Proof

In this section, we provide a proof for Theorem 4.1. We begin with a lemma that will be helpful for the proof:

Lemma A.1. Consider an individual x whose appearance is bound by $(\hat{\rho}, \hat{K})$ in front of a camera whose policy is (ρ, K, ϵ). For every query Q there exists $\alpha, \beta \in \mathbb{R}$ such that $\alpha K(1+\beta\rho) \leq \Delta_{(\rho,K)}(Q) \leq \alpha K(2+\beta\rho)$.

Proof. Any PRIVID query must contain some aggregation agg as the outer-most relation, and thus we can write $Q := \Pi_{agg}(R)$. $\Delta_{(\rho,K)}(Q)$ is defined in Figure 9 for five possible aggregation operators, which are each a function of $\Delta_{(\rho,K)}(R)$ (the sensitivity of their inner relation R).

First, we will prove these bounds are true for the inner relation $\Delta_{(\rho,K)}(R)$ by induction on R (all rules for $\Delta_{(\rho,K)}(R)$ given by Figure 9):

Case (Base): $R := t$ When R is an intermediate PRIVID table t, its sensitivity is given directly by Equation 5.2, where $\alpha = \mathsf{max\_rows}_t$ and $\beta = 1/c$. Note, the $(1 + \cdots)$ and $(2 + \cdots)$ in the lemma inequalities bound $\lceil \frac{\rho}{c} \rceil$.

Case (Selection): $R := \sigma(R')$. When R is a selection from R', $\Delta_{(\rho,K)}(R) = \Delta_{(\rho,K)}(R')$. If $\Delta_{(\rho,K)}(R')$ is bound by the inequalities in the lemma statement, then $\Delta_{(\rho,K)}(R)$ is too.

Case (Projection): $R := \Pi(R')$. Same as selection.

Case (GroupBy and Join): $R := \gamma(R_1 \bowtie ... \bowtie R_i)$ When R is a join of relations R_i proceeded by a GroupBy, $\Delta_{(\rho,K)}(R) = \sum_{i=1}^{N} \Delta_{(\rho,K)}(R_i)$. Let $\Delta_{(\rho,K)} R_i$ be parameterized by α_i and β_i. If each of $\Delta_{(\rho,K)}(R_i)$ are bound by the inequalities in the lemma, then $\sum_i \Delta_{(\rho,K)}(R_i)$ is as well, but with $\alpha = \sum_{i=1}^{N} \alpha_i$ and $\beta = \sum_{i=1}^{N} \beta_i$.

Finally, each of the supported aggregation operators only involve multiplying $\Delta_{(\rho,K)}(R)$ by constants (with respect to ρ and K), and thus these constants can be subsumed into α. □

We now restate Theorem 4.1 for the reader's convenience:

Theorem A.2. Consider a camera with a fixed policy (ρ, K, ϵ). If an individual x's appearance in front of the camera is bound by some $(\hat{\rho}, \hat{K})$, then PRIVID effectively protects x with $\hat{\epsilon}$-DP, where $\hat{\epsilon}$ is $O(\frac{\hat{\rho}\hat{K}}{\rho K})\epsilon$, which grows (degrades) as $(\hat{\rho}, \hat{K})$ increase while (ρ, K, ϵ) are fixed, and the constants do not depend on the query.

Proof. Recall from §5 that PRIVID uses the Laplace mechanism: it returns $Q(V) + \eta$ to the analyst, where $Q(V)$ is the raw query result, and $\eta \sim \mathsf{Laplace}(0, b)$, $b = \frac{\Delta_{(\rho,K)}(Q)}{\epsilon}$ and $\Delta_{(\rho,K)}(Q)$ is the global sensitivity of the query over any (ρ, K)-neighboring videos. Note that the sensitivity is purely a function of the query, and thus PRIVID samples noise using the same scale b regardless of how long any individual is actually visible in the video.
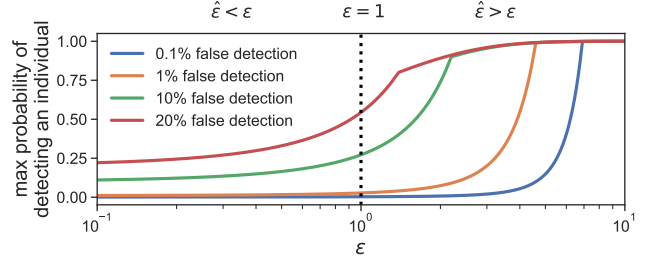
Figure 8: Plot of Equation A.4 for a few different levels of α. Note that the x-axis is plotted for absolute values of ϵ and is using a log scale. The y-axis is the maximum probability that an adversary with a given confidence level could detect whether or not x was present. If one draws a vertical line at the value of ϵ being enforced (e.g., we mark $\epsilon = 1$ here), the trend to the left shows how privacy is improved for individuals who are visible for less time, and the right shows how it degrades for those who are visible for more.

By Theorem B.2, this mechanism provides ϵ-DP for all (ρ, K)-bounded events. If we rearrange the equation for b so that $\epsilon = \frac{\Delta_{(\rho,K)}(Q)}{b}$, we can equivalently say that PRIVID guarantees $\frac{\Delta_{(\rho,K)}(Q)}{b}$-DP for all (ρ, K)-bounded events. Or, more generally, that a particular instantiation of PRIVID with policy $p = (\rho, K, \epsilon)$ guarantees $\hat{\epsilon}$-DP for all $(\hat{\rho}, \hat{K})$-bounded events in query Q, where [10]

$$\hat{\epsilon}_p(\hat{\rho}, \hat{K}, Q) = \frac{\Delta_{(\hat{\rho},\hat{K})}(Q)}{b} = \frac{\Delta_{(\hat{\rho},\hat{K})}(Q)}{\Delta_{(\rho,K)}(Q)/\epsilon} = \frac{\Delta_{(\hat{\rho},\hat{K})}(Q)}{\Delta_{(\rho,K)}(Q)}\epsilon$$

In other words, for a fixed policy, $\hat{\epsilon}$ defines the effective level of protection provided to an event as a function of the *event's* (not policy's) $(\hat{\rho}, \hat{K})$ bound.

From Lemma A.1, we can bound $\hat{\epsilon}$ as $\frac{\alpha\hat{K}(1+\beta\hat{\rho})}{\alpha K(2+\beta\rho)}\epsilon \leq \hat{\epsilon} \leq \frac{\alpha\hat{K}(2+\beta\hat{\rho})}{\alpha K(1+\beta\rho)}\epsilon$. To see where this comes from, note that $\hat{\epsilon}$ is minimized when the numerator is minimized (the lower bound from Lemma A.1) and the denominator is maximized (the upper bound from Lemma A.1). The same logic applies to the upper bound on $\hat{\epsilon}$.

We can simplify both bounds by first canceling α and then picking units of time such that $\beta = 1$ (β has dimensions of chunks per unit time). Thus,

$$\hat{\epsilon} \approx \frac{\hat{\rho}\hat{K}}{\rho K}\epsilon \tag{A.1}$$

□

A.2 Degradation of Privacy

Although $\hat{\epsilon}$ provides a way to quantify the level of privacy provided to each individual, it can be difficult to reason about relative values of ϵ and what they ultimately mean for privacy in practice. We can use the framework of binary hypothesis testing to develop a more intuitive understanding and ultimately visualize the degradation of privacy as a function of $\hat{\epsilon}$ relative to ϵ.

---

[10]Note the difference in subscript in the numerator and denominator.

Consider an adversary who wishes to determine whether or not some individual x appeared in a given video V. They submit a query Q to the system, and observe only the final result, A, which PRIVID computed as $A = Q(V) + \eta$, where η is a sample of Laplace noise as defined in the previous section. Based on this value, the adversary must distinguish between one of two hypotheses:

$$\mathcal{H}_0 : x \text{ does not appear in } V$$
$$\mathcal{H}_1 : x \text{ appears in } V$$

We write the false positive P_{FP} and false negative P_{FN} probabilities as:

$$P_{FP} = \mathbb{P}(x \in V | \mathcal{H}_0)$$
$$P_{FN} = \mathbb{P}(x \notin V | \mathcal{H}_1)$$

From Kairouz [46, Theorem 2.1], if an algorithm guarantees ϵ-differential privacy ($\delta = 0$), then these probabilities are related as follows:

$$P_{FP} + e^\epsilon P_{FN} \geq 1 \qquad (A.2)$$
$$P_{FN} + e^\epsilon P_{FP} \geq 1 \qquad (A.3)$$

Suppose the adversary is willing to accept a false positive threshold of $P_{FP} \leq \alpha$. In ther words, they will only accept $\mathcal{H}_1$ (x is present) if there is less than α probability that x is not actually present.

Rearranging equations A.2 and A.3 in terms of the probability of correctly detecting x is present ($1 - P_{FN}$), we have:

$$1 - P_{FN} \leq \qquad\qquad e^\epsilon P_{FP} \leq \qquad\qquad e^\epsilon \alpha$$
$$1 - P_{FN} \leq \quad e^{-\epsilon}(P_{FP} - (1 - e^\epsilon)) \leq \quad e^{-\epsilon}(\alpha - (1 - e^\epsilon))$$

Then, for a given threshold α, the probability that the adversary *correctly* decides x is present is *at most* the minimum of these:

$$\mathbb{P}(x \in V | \mathcal{H}_1) \leq \min\{e^\epsilon \alpha, e^{-\epsilon}(\alpha - (1 - e^\epsilon))\} \qquad (A.4)$$

In Fig. 8, we visualize A.4 as a function of ϵ for 4 different adversarial confidence levels (α=0.1%,1%,10%,20%). As an example of how to read this graph, suppose PRIVID uses a ($\rho = 60s, K = 1, \epsilon = 1$) policy ($\epsilon = 1$ marked with the dotted line). An individual who appears 3 times for $< 60s$ each is ($\rho = 60s, K = 3$)-bounded, and thus has an effective $\hat\epsilon = 3$ relative to the actual policy for most queries (Eq. A.1). If an adversary has a $\alpha = 1\%$ confidence level, then they would have at most a $\sim 20\%$ chance of correctly detecting the individual appeared, even though they appeared for far more than the policy allowed. We can also see that, for sufficiently small values of ϵ (e.g., $\epsilon < 1$), even if the adversary has a very liberal confidence level (say, 20%), a marginal increase in $\hat\epsilon$ relative to ϵ only gives the adversary a marginally larger probability of detection than they would have had otherwise.

An important takeaway is that, when an individual exceeds the (ρ, K) bound protected by PRIVID, their presence is not immediately revealed. Rather, as it exceeds the bound further, $\hat\epsilon$ increases, and it becomes more likely an adversary could detect the event.

B PRIVID Sensitivity Definition

Figure 9 provides the complete definition of sensitivity for a PRIVID query.

Lemma B.1. Given a relation R, the rules in Figure 9 are an upper bound on the global sensitivity of a (ρ, K)-bounded event in an intermediate table t.

Proof. Proof by induction on the structure of the query.
Case: t. $\Delta_\mathscr{P}(t)$ is given directly by Equation 5.2.
Case: $R' := \sigma_\theta(R)$. A selection may remove some rows from R, but it does not add any, or modify any existing ones, so in the worst case an individual can be in just as many rows in R' as in R and thus $\Delta_\mathscr{P}(R') \leq \Delta_\mathscr{P}(R)$ and the constraints remain the same. If θ includes a LIMIT $= x$ condition, then R' will contain at most x rows, regardless of the number of rows in R.
Case: $R' := \Pi_{a,\dots}(R)$. A projection never changes the number of rows, nor does it allow the data in one row to influence another row, so in the worst case an individual can be in at most the same number of rows in R' as in R ($\Delta_\mathscr{P}(R') \leq \Delta_\mathscr{P}(R)$) and the size constraint $\tilde{C}_s(R)$ remains the same. If the projection transforms an attribute by applying a stateless function f to it, then we can no longer many assumptions about the range of values in a ($\tilde{C}_r(R', a) = \varnothing$), but nothing else changes because the stateless nature of the function ensures that data in row cannot influence any others.
Case: GroupBy. A GROUP BY over a fixed set of a n keys is equivalent to n separate queries that use the same aggregation function over a $\sigma_{\text{WHERE}col=key}(R)$. If the column being grouped is a user-defined column, PRIVID requires that the analyst provide the keys directly. If the column being grouped is one of the two implicit columns (chunk or region), then the set of keys is not dependent on the contents of the data (only its length) and thus are fixed regardless.
Case: Join. Consider a query that computes the size of the intersection between two cameras, PROCESS'd into intermediate tables t_1 and t_2 respectively. If $\Delta(t_1) = x$ and $\Delta(t_2) = y$, it is tempting to assume $\Delta(t_1 \cap t_2) = \min(x, y)$, because a value needs to appear in both t_1 and t_2 to appear in the intersection. However, because the analyst's executable can populate the table arbitrarily, they can "prime" t_1 with values that would only appear in t_2, and vice versa. As a result, a value need only appear in either t_1 or t_2 to show up in the intersection, and thus $\Delta(t_1 \cap t_2) = x + y$ (the sum of the sensitivities of the tables). $\qquad\square$

Theorem B.2. Consider an adaptive sequence (§2.3) of n queries $Q_1, \dots, Q_n$, each over the same camera C, a privacy policy (ρ_C, K_C), and global budget ϵ_C. PRIVID (Algorithm 1) provides (ρ_C, K_C, ϵ_C)-privacy for all $Q_1, \dots, Q_n$.

Proof. Consider two queries Q_1 (over time interval I_1, using chunk size c_1 and budget ϵ_1) and Q_2 (over I_2, using c_2 and ϵ_2). Let $v_1 = V[I_1]$ be the segment of video Q_1 analyzes and $v_2 = V[I_2]$ for Q_2. Let E be a (ρ, K)-bounded event.

Notation	$\mathcal{P}$	Privacy policy for each camera: $\{(\rho, K)_c \; \forall \; c \in \text{cameras}\}$			
	$\Delta_{\mathcal{P}}(R)$	Maximum number of rows in relation R that could differ by the addition or removal of any (ρ, K)-bounded event.			
	$\vec{C}_r(R, a)$	Range constraint: range of attribute a in R			
	$\vec{C}_s(R)$	Size constraint: upper bound on total number of rows in R			
	$\varnothing$	Indicates that a relational operator leaves a constraint unbound. If this constraint is required for the aggregation, it must be bound by a predecessor. If it is not required, it can be left unbound.			

	Function	Definition	Constraints	Sensitivity ($\Delta(Q)$)
Aggregation Functions	Count	$Q := \Pi_{\text{count}(*)}(R)$	Δ	$1 \cdot \Delta(R)$
	Sum	$Q := \Pi_{\text{sum}(a)}(R)$	$\Delta, \vec{C}_r$	$\Delta(R) \cdot \vec{C}_r(R, a)$
	Average	$Q := \Pi_{\text{avg}(a)}(R)$	$\Delta, \vec{C}_r, \vec{C}_s$	$\frac{\Delta(R) \cdot \vec{C}_r(R,a)}{\vec{C}_s(R)}$
	Std. Dev	$Q := \Pi_{\text{stddev}(a)}(R)$	$\Delta, \vec{C}_r, \vec{C}_s$	$\Delta(R) \cdot \vec{C}_r(R,a)/\sqrt{\vec{C}_s(R)}$
	Argmax	$Q := \Pi_{\text{argmax}(a)}(R)$	$\Delta, a \in K$	$\max_{k \in K} \Delta(\sigma_{a=k}(R))$

Operator	Type	Definition	$\Delta_{\mathcal{P}}(R')$	$\vec{C}_r(R', a_i)$	$\vec{C}_s(R')$
Base Case	Base Table	R	$mr \cdot K \cdot (1 + \lceil \frac{\rho}{c} \rceil)$	$\varnothing$	$\varnothing$
Selection (σ)	Standard selection: rows from R that match WHERE condition	$R' := \sigma_{\text{WHERE}(\ldots)}(R)$	$\Delta_{\mathcal{P}}(R)$	$\vec{C}_r(R, a_i)$	$\vec{C}_s(R)$
	Limit: first x rows from R	$R' := \sigma_{\text{LIMIT}=x}(R)$	$\Delta_{\mathcal{P}}(R)$	$\vec{C}_r(R, a_i)$	$\min(x, \vec{C}_s(R))$
Projection (Π)	Standard projection: select attributes $a_i, \ldots$ from R	$R' := \Pi_{a_i, \ldots}$	$\Delta_{\mathcal{P}}(R)$	$\vec{C}_r(R, a_i)$	$\vec{C}_s(R)$
	Apply (user-provided, but stateless) f to column a_i	$R' := \Pi_{f(a_i), \ldots}$	$\Delta_{\mathcal{P}}(R)$	$\varnothing$	$\vec{C}_s(R)$
	Add range constraint to column a_i	$R' := \Pi_{a_i \in [l_i, u_i], \ldots}$	$\Delta_{\mathcal{P}}(R)$	$[l_j, u_j]$ if $a_i \neq \varnothing$ $\vec{C}_r(R, a_i)$ otherwise	$\vec{C}_s(R)$
GroupBy (γ)	Group attribute(s) (g_i) are chunk (or binned chunk) or region	$R' := g_j, \ldots \gamma_{\text{agg}(a_i)}, \ldots$ $g_j := \text{chunk} \| \text{bin}(\text{chunk})$	Equation 5.2	$\Delta(\text{agg}(a_i))$	$\frac{\vec{C}_s(R)}{(\text{bin size})}$
	Group attribute(s) (g_j) are *not* chunk or region	$R' := g_j, \ldots \gamma_{\text{agg}(a_i)}, \ldots$	$\Delta_{\mathcal{P}}(R)$	$\varnothing$	$\varnothing$
	... discrete set of keys provided for each group (constrains size)	$R' := g_j \in K_j, \ldots \gamma_{\text{agg}(a_i)}, \ldots$	...	...	$\Pi_j \|K_j\|$
	... aggregation constrains range: $agg(a_i) \in [l_i, u_i]$	$R' := g_j, \ldots \gamma_{\text{agg}(a_i) \in [l_i, u_i]}, \ldots$	...	$[l_j, u_j]$ if $a_i \neq \varnothing$ $\vec{C}_r(R, a_i)$ otherwise	...
Joins* ($\bowtie$)	*When *immediately* preceded by GroupBy *over the same key(s)*	$R' := {}_g\gamma_{\text{agg}(a)}(R_1 \bowtie_g \cdots \bowtie_g R_n)$	$\sum_{i=1}^n \Delta_{\mathcal{P}}(R_i)$	(GroupBy rules)	(GroupBy rules)
	... equijoin on g_j (intersection on g_j)	$R' := {}_g\gamma_{\text{agg}(a)}(R_1 \bowtie_g \cdots \bowtie_g R_n)$			
	... outer join on g_j (union on g_j)				

Figure 9: Full set of rules for PRIVID's sensitivity calculation.

Case 1: I_1 and I_2 are not ρ-disjoint The budget check (lines 1-3 in Algorithm 1) ensures that these two queries must draw from the same privacy budget, because their effective ranges overlap by at least one frame (but may overlap up to all frames). By Theorem 5.1, PRIVID is (ρ, K, ϵ_1)-private for Q_1 and (ρ, K, ϵ_2)-private for Q_2. By Dwork [37, Theorem 3.14], the combination of Q_1 and Q_2 is $(\rho, K, \epsilon_1 + \epsilon_2)$-private.

Case 2: I_1 and I_2 are ρ-disjoint In other words, $I_1 + \rho < I_2 - \rho$, thus the budget check (lines 1-3) allows these two queries to draw from entirely separate privacy budgets. Since the intervals are ρ-disjoint, and all segments in E must have duration $\leq \rho$, it is not possible for the same segment to appear in even a single frame of *both* intervals.

Let K_1 be the number of segments contained in I_1, each of duration $\leq \rho$, and K_2 be the remaining segments contained in I_2, each of duration $\leq \rho$. In other words, E is (ρ, K_1)-bounded in v_1 and (ρ, K_2)-bounded in v_2. Since E has at most K segments, $K_1 + K_2 \leq K$. We need to show that the probability of observing both A_1 and A_2 if the inputs are the actual segments v_1 and v_2 is close (e^ϵ) to the probability of observing those values if the inputs are the neighboring segments v_1' and v_2':

$$\frac{\Pr[A_1 = Q_1(v_1), A_2 = Q_2(v_2)]}{\Pr[A_1 = Q_1(v_1'), A_2 = Q_2(v_2')]} \leq \exp(e)$$

Since the probability of observing A_1 is independent of observing A_2 (randomness is purely over the noise added by PRIVID):

$$\frac{\Pr[A_1 = Q_1(v_1), A_2 = Q_2(v_2)]}{\Pr[A_1 = Q_1(v_1'), A_2 = Q_2(v_2')]}$$

$$\leq \frac{\Pr[A_1 = Q_1(v_1)]\Pr[A_2 = Q_2(v_2)]}{\Pr[A_1 = Q_1(v_1')]\Pr[A_2 = Q_2(v_2')]}$$

$$\leq \frac{\frac{1}{2b_1}\exp(-\frac{|A_1 - Q_1(v_1)|}{b_1})\frac{1}{2b_2}\exp(-\frac{|A_2 - Q_2(v_2)|}{b_2})}{\frac{1}{2b_1}\exp(-\frac{|A_1 - Q_1(v_1')|}{b_1})\frac{1}{2b_2}\exp(-\frac{|A_2 - Q_2(v_2')|}{b_2})}$$

(By Algorithm 1, Line 13)

$$= \exp(\frac{|A_1 - Q_1(v_1')| - |A_1 - Q_1(v_1)|}{b_1} + \frac{|A_2 - Q_2(v_2')| - |A_2 - Q_2(v_2)|}{b_2})$$

If K_1 segments are in v_1 and K_2 segments are in v_2, the numerator of each fraction above is the sensitivity of a (ρ, K_1)-bounded event and a (ρ, K_2)-bounded event, respectively. b_1 and b_2 are the amount of noise actually added to the query, which are both based on K:

$$\leq \exp(\frac{\Delta_{(\rho, K_1)}(Q_1)}{\Delta_{(\rho, K)}(Q_1)/\epsilon} + \frac{\Delta_{(\rho, K_2)}(Q_2)}{\Delta_{(\rho, K)}(Q_2)/\epsilon})$$

$$= \exp(\epsilon \cdot (\frac{K_1(\lceil \frac{\rho}{c_1} \rceil + 1)}{K(\lceil \frac{\rho}{c_1} \rceil + 1)} + \frac{K_2(\lceil \frac{\rho}{c_2} \rceil + 1)}{K(\lceil \frac{\rho}{c_2} \rceil + 1)}))$$

(by Equation 5.2)

$$= \exp(\epsilon \cdot (\frac{K_1}{K} + \frac{K_2}{K})) \quad \text{(recall } K \geq K_1 + K_2\text{)}$$

$$\leq \exp(\epsilon)$$

$\square$

C Query Details

C.1 Case 1 Query Statements

```
Case 1: Query 1

SPLIT campus
    BEGIN 06-01-2019/06:00am END 06-01-2019/06:00pm
    BY TIME 30sec STRIDE 0sec
    BY REGION
    WITH MASK C1
    INTO campusChunks;
PROCESS campusChunks USING count_ppl_campus.py TIMEOUT 1sec
    PRODUCING 1 ROWS
    WITH SCHEMA (ppl:NUMBER=0)
    INTO campusTable;
SELECT hour,sum(RANGE(ppl,0,6)) from campusTable
    GROUP BY hour
    CONSUMING eps=1.0;
```

```
Case 1: Query 2

SPLIT highway
    BEGIN 06-01-2019/06:00am END 06-01-2019/06:00pm
    BY TIME 30sec STRIDE 0sec
    BY REGION
    WITH MASK H2
    INTO highwayChunks;
PROCESS highwayChunks USING count_cars.py TIMEOUT 1sec
    PRODUCING 1 ROWS
    WITH SCHEMA (cars:NUMBER=0)
    INTO highwayTable;
SELECT hour, sum(RANGE(cars,0,100)) from highwayTable
    GROUP BY hour
    CONSUMING eps=1.0;
```

```
Case 1: Query 3

SPLIT urban
    BEGIN 06-01-2019/06:00am END 06-01-2019/06:00pm
    BY TIME 30sec STRIDE 0sec
    BY REGION
    WITH MASK U2
    INTO urbanChunks;
PROCESS campusChunks USING count_ppl_urban.py TIMEOUT 1sec
    PRODUCING 1 ROWS
    WITH SCHEMA (ppl:NUMBER=0)
    INTO campusTable;
SELECT hour, sum(RANGE(ppl,0,23)) from campusTable
    GROUP BY hour
    CONSUMING eps=1.0;
```

C.2 Case 2: Complex Sensitivity Example

The code block for Case 2 contains Queries 4-6, which are computed over the same set of intermediate tables.

To demonstrate the sensitivity computation for a complex PRIVID query, we focus on Query 4. This query aims to estimate the typical working hours of taxis in the city of Porto, Portugal; it first computes the difference between the first and last time each taxi (identified by plate) was seen (by either camera 10 or 27) on a given day, then averages across all taxis and days (over a year).

In order to ensure all variables needed for the aggregation are properly constrained, we make two assumptions: most taxis will not work more than 16 hours in a day, and there are roughly 300 public taxis in Porto (based on public information). We can express this query in relational algebra as follows:

$$\Pi_{\text{Avg(hrs)}}\left(\sigma_{\text{limit(plates)}=300}\left(_{\text{plate,day}}\gamma_{\text{range(chunks)}\in[0,16]}(t_1\cup t_2)\right)\right)$$

```
Case 2: Queries 4-6

-- Repeat for portoCam1...portoCam127:
SPLIT portoCam1
    BEGIN 07-01-2013/12:00am END 07-01-2014/12:00am
    BY TIME 15sec STRIDE 0sec
    INTO chunks1;
-- Repeat for chunks1...chunks127:
PROCESS chunks1 USING porto.py TIMEOUT 1sec
    PRODUCING 3 ROWS
    WITH SCHEMA (plate:STRING="")
    INTO table1;

-- Query 4: Average Taxi Working Hours
SELECT avg(avg_shift) FROM
    (SELECT plate,avg(RANGE(shift, [0,16])) FROM
        (SELECT plate,day,(max(chunk)-min(chunk) as shift) FROM
            table10 UNION table27 GROUP BY plate,day(chunk))
        GROUP BY plate LIMIT 300)
CONSUMING eps=0.33;
-- Query 5: # Taxis Traversing Both Locations On Same Day
SELECT day,count(DISTINCT plate) FROM
    (SELECT day,plate FROM
        table10 INNER JOIN table27 ON
        (table10.plate=table27.plate AND table10.day=table27.day)
    )
    GROUP BY day
CONSUMING eps=0.33;
-- Query 6: Camera with highest daily traffic
SELECT argmax(arg=cam, target=avg_daily) FROM
  (SELECT "cam1" as cam, avg(daily) as avg_daily FROM
    (SELECT day,count(DISTINCT plate) as daily FROM
      table1 GROUP BY day))
  UNION
  // ...
  UNION
  (SELECT "cam127" as cam, avg(daily) as avg_daily FROM ...)
CONSUMING eps=0.33;
```

We use the policy $\mathcal{P} = \{(\rho = 45s, K = 1)_{c_1}, (195s, 1)_{c_2}\}$ (the max observed persistence over historical data for each camera) and an ϵ of 1.

First, we compute the base sensitivity of each table. The SPLIT statement specifies the video will be split into 15 second chunks with 0 stride, and that each chunk will produce a maximum of 3 rows. With this we can compute: $\Delta_{\mathcal{P}}(t_1) = \lceil\frac{(45*\text{fps}-1)}{15*\text{fps}}\rceil + 1 = 4 \cdot 3 = 12$ and $\Delta_{\mathcal{P}}(t_2) = \lceil\frac{195*\text{fps}-1}{15*\text{fps}}\rceil + 1 = 14 \cdot 3 = 42$. When we combine them with a union, their sensitivities add: $\Delta_{\mathcal{P}}(t_1 \cup t_2) = 12 + 42 = 54$. The GROUP BY creates a new table with a row per plate per day, and constrains the range of the aggregate value shift to $[0, 16]$ (range(a,b) returns $|b - a|$, i.e., the time between the first and last appearance of a taxi on a given day), but we don't know how many unique plates there might be, so the size $\tilde{C}_s(\gamma(...))$ is unconstrained. We add σ_{limit} to manually enforce a maximum of 300 plates per day, which gives us a constraint $\tilde{C}_s(\sigma(...)) = 300\text{plates} * 365\text{days} = 109,500$. We now have all the constraints necessary to compute the sensitivity of the average aggregation: $\Delta_{\mathcal{P}}^{\text{AVG}}(R) = \frac{\Delta_{\mathcal{P}}(R)\tilde{C}_r(R,\text{shift},)}{\tilde{C}_s(R)} = \frac{54\cdot16}{109,500} = 0.0079$. Since PRIVID uses the Laplace mechanism to add noise, we can use the inverse CDF of the Laplace distribution to bound the expected error based on Δ with a given confidence level. For example, $\mathcal{L}^{-1}(p = 0.999, u = 0, b = \frac{\Delta}{\epsilon} = \frac{0.0079}{0.33}) \leq 0.15$ hours with 99.9% confidence.

Spectrum: High-bandwidth Anonymous Broadcast

Zachary Newman
MIT CSAIL
zjn@mit.edu

Sacha Servan-Schreiber
MIT CSAIL
3s@mit.edu

Srinivas Devadas
MIT CSAIL
devadas@csail.mit.edu

Abstract

We present Spectrum, a high-bandwidth, metadata-private file broadcasting system. In Spectrum, a small number of broadcasters share a file with many subscribers via two or more non-colluding broadcast servers. Subscribers generate cover traffic by sending dummy files, hiding which users are broadcasters and which users are only consumers.

Spectrum optimizes for a setting with few broadcasters and many subscribers—as is common to many real-world applications—to drastically improve throughput over prior work. Malicious clients are prevented from disrupting broadcasts using a novel blind access control technique that allows servers to reject malformed requests. Spectrum also prevents deanonymization of broadcasters by malicious servers deviating from protocol. Our techniques for providing malicious security are applicable to other systems for anonymous broadcast and may be of independent interest.

We implement and evaluate Spectrum. Compared to the state-of-the-art in cryptographic anonymous communication systems, Spectrum's peak throughput is 4–120,000× faster (and commensurately cheaper) in a broadcast setting. Deployed on two commodity servers, Spectrum allows broadcasters to share 1 GB (two full-length 720p documentary movies) in 13h 20m with an anonymity set of 10,000 (for a total cost of about $6.84). These costs scale roughly linearly in the size of the file and total number of users, and Spectrum parallelizes trivially with more hardware.

1 Introduction

An informed public often depends on whistleblowers, who expose misdeeds and corruption. Over the last century, whistleblowers have exposed financial crimes, government corruption [61, 69, 75], risks to public health [43, 52], Russian interference in the 2016 U.S. presidential election [61, 70], presidential misconduct [17, 45, 67, 79], war and human rights crimes [5, 38, 87], and digital mass surveillance by U.S. government agencies [18]. Philosophers debate whistleblowing ethics [3, 35], but agree it often has positive effects.

Motivation for this work. Whistleblowers take on great personal risks in bringing misdeeds to light. The luckiest enjoy legal protections [88] or financial reward [89]. But many face exile [18], incarceration [50, 70, 74], or risk their lives [87]. More recently, political activist Alexei Navalny was detained and sentenced to prison following the release of documents accusing Russian president Vladimir Putin of corruption and embezzlement [80].

To mitigate these risks, many whistleblowers turn to technology to protect themselves [47]. Secure messaging apps Signal [26] and SecureDrop [8] have proven to be an important resource to whistleblowers and journalists [44, 84]. Encryption does its job, even against the NSA [92]—but it may not be enough to protect from powerful adversaries.

Since the Snowden revelations, governments and the press have focused on *metadata*. The source, destination, and timing of encrypted data can leak information about its contents. For instance, prosecutors used SFTP metadata in the case against Chelsea Manning [96]. Newer technology is still vulnerable: a federal judge found Natalie Edwards guilty on the evidence of metadata from an encrypted messaging app [50]. To protect whistleblowers and protect against powerful adversaries (e.g., corrupted internet service providers), systems must be designed with metadata privacy in mind.

Many academic and practical metadata-hiding systems provide solutions to this problem for some applications. Tor [37] boasts a distributed network of 6,000 nodes and 2 million daily active users (the only such system with wide usage). Tor is fast enough for web browsing, but deanonymization attacks identify users with a high success rate based on observed traffic [9, 14, 42, 48, 53, 65, 68]. Moreover, the effectiveness of deanonymization attacks increases with the size of the traffic pattern. Whistleblowers using Tor to upload large files can be more easily deanonymized for this reason.

Some recent academic research systems [2, 30, 41, 54–56, 58, 86, 90] address the problem of hiding metadata in anonymous communication, providing precise security guarantees for both direct messaging and "Twitter"-like broadcast applications. However, a limitation of all existing systems is

that they are designed for low-bandwidth content, incurring impractical latencies with large messages (see Section 8).

Contributions. Motivated by the lack of anonymous broadcast systems suitable for high-throughput data dumps, we design and build Spectrum: a system for high-bandwidth metadata-private anonymous broadcasting. Spectrum is the first anonymous broadcast system supporting high-bandwidth, many-user settings. It optimizes for the many-subscriber and few-broadcaster setting, which reflects the real-world usage of broadcast platforms. Per-request, Spectrum's server-side processing costs grow with the number of broadcasters rather than the total number of users, significantly improving performance over prior work when only a subset are broadcasting.

This paper contributes:

1. Design and security analysis of Spectrum, a system for high-bandwidth broadcasting with strong robustness and privacy guarantees against malicious adversaries,

2. A notion of blind access control for anonymous communication, along with a construction and a black-box transformation to efficiently support large (1 GB) messages,

3. Identification of an "audit attack" that allows malicious servers to deanonymize users, and BlameGame, a black-box blame protocol to "upgrade" Spectrum and similar systems to defend against this attack.

4. An open-source implementation of Spectrum, evaluated in comparison to other anonymous communication systems. We show that Spectrum supports high-bandwidth, latency-sensitive applications such as real-time anonymous podcasting, video streaming, and large file leaks.

Limitations. Like other metadata-private systems, Spectrum provides anonymity among honest online users and requires all users to contribute cover messages to a broadcast (to perfectly hide network metadata). Additionally, Spectrum achieves peak performance with exactly two servers (similarly to Riposte and Express [30, 41]). Instantiating with more than two servers requires using a less (concretely) efficient cryptographic primitive: a seed-homomorphic pseudorandom generator [12]. Finally, Spectrum requires a one-time "bootstrapping" process at setup time (similar to other systems [4, 29, 41, 58, 90, 95]); see Section 2.3.

2 Anonymous broadcast

In this section, we describe anonymous broadcast and its challenges, along with our system design and techniques.

Setting and terminology. In anonymous broadcast, one or more users/clients (*broadcasters*) share a *message* (e.g., file) while preventing network observers from learning its source. In Spectrum, passive users generate cover traffic (dummy messages) to increase the size of the *anonymity set* (the set of users who could have plausibly sent the broadcast message).

These passive users are *subscribers*, consuming broadcasts. Because the message sources are anonymous, the servers publish distinct messages to different *channels* or slots. Every broadcaster has exactly one channel, which they anonymously publish to in every iteration of the protocol. The servers cannot distinguish between a subscriber sending cover traffic and a broadcaster writing to a channel.

The primary challenge in anonymous broadcasting is preventing *disruption* by malicious clients: in simple broadcasting systems, users can clobber other users' messages via undetectable deviations from the protocol [2, 30, 41]. We first begin by explaining the standard building-block for achieving anonymous broadcasting [2, 30]. In subsequent sections, we build off of this basic scheme to achieve disruption resistance.

2.1 DC-nets

Chaum [23] presents DC-nets, which enable a rudimentary form of anonymous broadcast. DC-nets use secret-sharing to obscure the source of data in the network. Like prior work [2, 30, 41, 95], we instantiate a DC-net with two or more servers and many clients. One client (the broadcaster) wishes to share a file; all other clients (subscribers) provide cover traffic. In a two-server DC-net, the ith client samples a random bit string r_i and sends $r_i \oplus m_i$ to ServerA and r_i to ServerB. Servers can recover m_i by combining their respective shares:

$$m_i = (m_i \oplus r_i) \oplus (r_i).$$

If exactly one of N clients shares a message $m_i = \widehat{m}$ while all other clients share $m_i = 0$, the servers can recover $\widehat{m}$ (without learning which client sent $m_i = \widehat{m}$) by first locally aggregating all received shares as $\mathsf{agg}_A = \bigoplus_i (r_i \oplus m_i)$ and $\mathsf{agg}_B = \bigoplus_i r_i$ and then revealing the aggregation to the other server.

Because all subscribers send shares of zero, combining the local aggregations recovers the broadcaster's message:

$$\widehat{m} = \mathsf{agg}_A \oplus \mathsf{agg}_B.$$

The above scheme protects client anonymity, as each server sees a uniformly random share from each client.

DC-net challenges. While DC-nets allow fast anonymous broadcast, users can undetectably *disrupt* the broadcast by sending non-zero shares. Preventing such disruptions is a major challenge and primary source of latency in prior DC-net-based systems [2, 29, 30, 41, 54, 55, 95] (see related work; Section 8). Also, while DC-nets enable one broadcaster to transmit a message, many clients may wish to broadcast. Repeating the protocol in parallel is inefficient, requiring bandwidth linear in the number of broadcasters. Even prior works which overcome the linear (in the number of broadcasters) bandwidth overhead of naïve protocol repetition suffer from linear server-side work per client, regardless of whether or not all clients are broadcasters. In Spectrum, the bandwidth
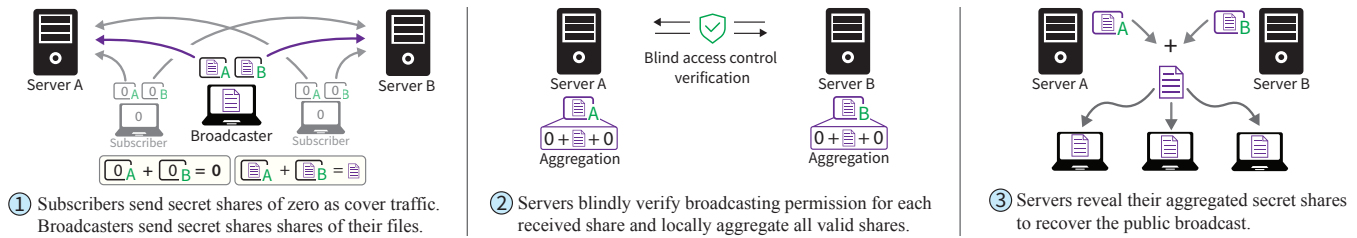
Figure 1: High-level overview of Spectrum when instantiated with two servers (and one broadcaster).

Panel 1: Subscribers send secret shares of zero as cover traffic. Broadcasters send secret shares shares of their files.

Panel 2: Servers blindly verify broadcasting permission for each received share and locally aggregate all valid shares.

Panel 3: Servers reveal their aggregated secret shares to recover the public broadcast.

overhead grows *logarithmically* in the number of broadcasters. Additionally, the server-side work only grows linearly in the number of broadcasters, rather than the total number of clients. We compare this work and other DC-net-based anonymous broadcasting systems in Table 1.

2.2 Main ideas in realizing Spectrum

Spectrum builds on top of DC-nets, improving efficiency and preventing disruption by malicious clients.

Practical efficiency. Spectrum capitalizes on the asymmetry of real-world broadcasting: there are typically fewer broadcasters than there are subscribers. While some prior works repeat many executions of the DC-net protocol more efficiently than the naïve scheme, they still reserve space (i.e., channels) for *every* client [2, 30]. As a consequence, the per-request computation on each server is linear in the number of clients, leading to high latency and "wasted" work. Spectrum derives anonymity from *all* clients, but only the total number of *broadcasters* influences the per-request work on each server (rather than the total number of clients in the system).

Preventing disruption. In Spectrum, we prevent broadcast disruption by developing a new idea: *anonymous* access control (Section 3.1), which we realize from the Carter-Wegman MAC [94]. We check access to each "channel" to ensure that only a user with a "broadcast key" can write to that channel.

Preventing "audit" attacks. Anonymous broadcast servers can *covertly* exclude a client in order to deanonymize the corresponding user. While vanilla DC-nets do not have this problem, prior anonymous broadcast systems leave out a client's share if they are found to be ill-formed. This is done to defend against disruption. However, it also makes it possible for a malicious server to exclude a user by framing them as malicious. In the broadcast setting, excluding a user can effectively deanonymize them. Abraham et al. [2] make the same observation and defend against the attack by requiring an honest-majority out of five or more servers. In Dissent [29], deanonymization is prevented with an expensive, after-the-fact blame protocol. Other systems [30, 41] are vulnerable to this attack (see Appendix A for details). Spectrum is the first system to efficiently defend against this attack while still preventing disruption per request (rather than assigning blame

after-the-fact). We achieve this by introducing BlameGame (Section 4.3), a lightweight blame protocol which can also be applied to other systems (e.g., Riposte [30] and Express [41]).

2.3 System overview

Spectrum is built using two or more broadcast servers (only one must be honest to guarantee anonymity; see Section 2.4) and many clients consisting of broadcasters and subscribers. One or more broadcaster(s) wish to share a message (as in the DC-net example). The subscribers generate cover traffic to increase the anonymity set. Each broadcaster has a private *channel*—or slot—for their message. Subscribers do not have channels. At the end of each round, Spectrum publishes the contents of each channel, hiding which client wrote to which channel (if any). Spectrum has three phases.

Setup. During setup, all broadcasters register with the servers. All users perform a setup-free anonymous broadcast protocol to establish a channel in Spectrum. Specifically, each broadcaster shares a public authentication key with the servers, which will be used to enforce anonymous access to write to a channel. At the end of the setup phase, the servers publish all parameters, including the number of channels and the maximum size of each broadcast message per round.

Main protocol. The protocol proceeds in one or more rounds (overview in Figure 1; details in Section 4.2). In each round, every client sends request shares to each server. The broadcasters send shares of their messages while the subscribers send empty shares. To enforce access control, the servers perform an efficient audit over the received shares: they obliviously check that each writer to a channel knows the secret channel broadcast key, *or* their message is zero. If the message shares pass the audit, the servers aggregate them as in a DC-net (Section 2.1). Otherwise, the servers perform a blame protocol (see BlameGame, summarized below). Finally, the servers combine aggregated shares to recover the messages.

BlameGame. If any client's request fails the audit, the servers perform BlameGame, a simple blame protocol (detailed in Section 4.3). BlameGame determines whether a client failed the access control check or if a server tampered with the client request in an attempt to frame a client as malicious. If the client is blamed, the servers drop the client's request and

proceed with the main Spectrum protocol. Otherwise, if a server is blamed, the honest server(s) abort. This protocol is much faster than fully aggregating a client's request, so a malicious client cannot use this to cause significant delays.

2.4 Threat model and security guarantees

Spectrum is instantiated with two (or more) broadcast servers and many clients (broadcasters and subscribers). Clients send shares of a message to the servers for aggregation.

Threat model

- No client is trusted by any honest server.
- Clients may deviate from the protocol, collude with other clients, or collude with a subset of malicious servers.
- At least one server must be honest to guarantee anonymity for clients (it does not matter which server is honest).
- Any subset of servers may deviate from the protocol and collude with malicious clients or the network adversary.

Assumptions. We make black-box use of public key infrastructure (e.g., TLS [73]) to encrypt data between clients and servers. We make the following cryptographic assumptions: (1) the hardness of the discrete logarithm problem [11], (2) the hardness of the decision Diffie-Hellman problem [10, 40] (when instantiated with more than two servers), and (3) the existence of hash functions and pseudorandom generators. We also assume a setup-free anonymous broadcast system [2, 30, 55, 95] for bootstrapping. As with prior work [2, 29, 30, 41, 55], we assume all communication between parties is observed by the network adversary.

Guarantees. Under the above threat model and assumptions, we obtain the following guarantees.

- *Anonymity.* An adversary controlling the network and a strict subset of servers and clients cannot distinguish between honest clients: broadcasters and subscribers are cryptographically indistinguishable in Spectrum. That is, no adversary observing the network and controlling a subset of servers and clients can distinguish between an honest subscriber and an honest broadcaster.
- *Availability.* If all servers follow the protocol, the system remains available (even if many clients are malicious). If any server halts or deviates from the protocol, then availability is not guaranteed and the protocol may abort.

Non-goals. We do not protect against denial-of-service attacks by a large number of clients (but we note that standard techniques, such as CAPTCHA [91], anonymous one-time-use tokens [33], or proof-of-work [39, 51] apply). Like all anonymous broadcast systems, intersection attacks on participation in the protocol can identify users, so Spectrum requires that users stay online for the duration of the protocol.

3 Spectrum with one channel

In this section, we introduce Spectrum with a single broadcaster (and therefore a single channel), two servers, and many subscribers. Figure 1 depicts an example. This setup mirrors the simplest DC-net protocol of Section 2.1. In Section 4, we extend Spectrum to many broadcasters and many servers.

3.1 Preventing disruption

We denote by $\mathbb{F}$ any finite field of prime order (e.g., integers mod p). We assume that all messages are elements in $\mathbb{F}$. (Section 5.1 shows how to efficiently support large binary messages in $\mathbb{F}_{2^\ell}$.) Each server receives secret-shares of a message m_i, where $m_i = 0 \in \mathbb{F}$ for subscribers and $m_i = \widehat{m} \in \mathbb{F}$ for the broadcaster. To prevent disruption, we enforce the following rule: for each channel, the broadcaster (with knowledge of a pre-established broadcast key) can send a non-zero message; all subscribers (who do not have the broadcast key) can only share a zero message. We give a new technique enabling the servers to verify the rule efficiently *without* learning anything except for the validity of the provided secret-shares.

New tool: anonymous access control. We adapt the Carter-Wegman MAC [21, 94] to provide a secret-shared "access proof" accompanying the message shares. Each client sends a secret-shared proof that it is either: (1) sending a share of a broadcast message with knowledge of the broadcast key; or (2) sending a cover message (i.e., $m_i = 0$) that does not affect the final aggregate computed by the servers.

Carter-Wegman MAC. Let $\mathbb{F}$ be any finite field of sufficiently large size for security. Sample a random authentication key $(\alpha, \gamma) \in \mathbb{F} \times \mathbb{F}$ and define $\mathsf{MAC}_{(\alpha,\gamma)}(m) = \alpha \cdot m + \gamma \in \mathbb{F}$. Observe that $\mathsf{MAC}_{(\alpha,\gamma)}$ is a *linear function* of the message, which makes it possible to verify a *secret-shared tag* for a *secret-shared* message. We demonstrate this with two servers ServerA and ServerB. Let $t = \mathsf{MAC}_{(\alpha,\gamma)}(m)$. If m is additively secret-shared as $m = m_A + m_B \in \mathbb{F}$, and t is secret shared as $t = t_A + t_B \in \mathbb{F}$, the servers (knowing α and γ) can verify that the tag corresponds to the secret-shared message:

- ServerA computes $\beta_A \leftarrow (\alpha \cdot m_A - t_A) \in \mathbb{F}$.
- ServerB computes $\beta_B \leftarrow (\alpha \cdot m_B - t_B) \in \mathbb{F}$.
- Servers swap β_A and β_B and check if $\beta_A + \beta_B = \gamma \in \mathbb{F}$.

The final condition only holds for a valid tag. Neither server learns anything about the message m in the process (apart from the tag validity) since both the message and tag remain secret-shared between servers.

If both the servers and the broadcaster know the key (α, γ), the broadcaster can compute a tag t which the servers can check for correctness as above. However, there are two immediate problems to resolve. First, subscribers cannot generate valid tags on zero messages without knowledge of (α, γ). Second, an honest-but-curious (or compromised) server can share (α, γ) with a malicious client who can then covertly

	Request Size	Audit Size	Audit Rounds	Server Work	Malicious Security	Disruption Handling	Blame Protocol	Comments
Blinder [2]	$\|m\| \cdot \sqrt{N}$	$\lambda \cdot \|m\|$	$\log N$	$N \cdot \|m\|$	✓	Prevent	N/A	Requires 5+ servers and MPC
Dissent [29]	$\|m\| \cdot L + N$	N/A	N/A	$L \cdot \|m\|$	✓	Detect	Expensive	Blame quadratic in N
PriFi [6]	$\|m\| \cdot L + N$	N/A	N/A	$L \cdot \|m\|$	✓	Detect	Expensive	Similar to Dissent
Riposte [30]	$\|m\| + \sqrt{N}$	$\sqrt{N}$	1	$N \cdot \|m\|$	✗	Prevent	None	Requires a separate audit server
Express [41]	$\|m\| + \log L$	λ	1	$L \cdot \|m\|$	✗	Prevent	None	Exactly 2 servers
Two-Server	$\|m\| + \log L$	λ	1	$L \cdot \|m\|$	✓	Prevent	Lightweight	With tree-based DPF [15]
Multi-Server	$\|m\| + \sqrt{L}$	λ	1	$L \cdot \|m\|$	✓	Prevent	Lightweight	With seed-homomorphic DPF [12, 30]

Table 1: Per-request asymptotic efficiency of Spectrum (highlighted) and prior anonymous broadcasting systems for L broadcasters, N total users, $|m|$-sized messages, and global security parameter λ. $O(\cdot)$ notation suppressed for clarity. Spectrum's advantages include: a request size that is sublinear in L (Section 5.1) and independent of N (Section 3.3), a protocol for lightweight auditing of client requests to prevent disruption (Section 3.1), and a fast blame protocol for security against malicious servers (Section 4.3).

disrupt a broadcast. (A malicious server can always *overtly* disrupt the broadcast by refusing to participate in Spectrum.)

Allowing forgeries on zero messages. To allow subscribers to send the zero message *without* knowing the secret MAC key, we leverage the following insight from the SPDZ [31] multi-party computation protocol. The γ value acts solely as a "nonce" to prevent forgeries on the message $0 \in \mathbb{F}$ [93]. Because of this, we can eliminate γ while still having the desired unforgeability property of the original MAC for all *non-zero* messages. When evaluated over secret shares, $\mathsf{MAC}_\alpha(m) = \alpha \cdot m \in \mathbb{F}$ maintains security for all $m \neq 0$. This satisfies our requirement: Subscribers can send $m = 0$ and a valid tag $t = 0$ *without* knowing α (i.e., subscribers can "forge" a valid tag but only for $m = 0$).

Preventing client-server collusion. To prevent an honest-but-curious server from collaborating with a malicious client to disrupt a broadcast, we must prevent the servers from knowing the broadcast key α while still allowing them to check the MAC tag. To achieve this, we shift the entire verification procedure "to the exponent" of a group $\mathbb{G}$ of prime order p (so that the exponent constitutes a field $\mathbb{F}_p$). For security, we also require that the discrete logarithm problem is computationally intractable in the group $\mathbb{G}$ [85]. Then, instead of α, the servers obtain a public verification key g^α (here g is a generator of $\mathbb{G}$) from each broadcaster. All verification proceeds as before. Each client generates secret-shares (t_A, t_B) of a tag t and shares (m_A, m_B) of the message m, which are distributed to the servers.

- ServerA computes $g^{\beta_A} \leftarrow (g^\alpha)^{m_A} / g^{t_A}$.
- ServerB computes $\beta_B \leftarrow (g^\alpha)^{m_B} / g^{t_B}$.
- Servers swap g^{β_A} and g^{β_B} and check if $g^{\beta_A} \cdot g^{\beta_B} = g^0 = 1_\mathbb{G}$.

Security. The unforgeability properties are inherited from the Carter-Wegman MAC. Client anonymity (i.e., secrecy of the message m_i) follows from the additive secret-sharing.

Client-server collusion is prevented by only the broadcaster knowing the broadcast key α. See Section 6 for full analysis.

3.2 Putting things together

In this section, we combine DC-nets for broadcast with anonymous access control to realize Spectrum with a single channel, generalizing to multiple channels in Section 4.

Setup: broadcast key distribution. The setup in Spectrum involves the broadcaster anonymously "registering" with the servers by giving them the authentication public key g^α. The servers must not learn the identity of the broadcaster when receiving this key, which leads us to a somewhat circular problem: broadcasters need to anonymously broadcast a key in order to broadcast anonymously. We solve this one-time setup problem as follows. All clients use a slower anonymous broadcast system suitable for low-bandwidth content at system setup time [2, 30, 55, 95]. The broadcaster shares an authentication key while subscribers share nothing. Keys are small (e.g., 64 bytes) and therefore practical to share with existing anonymity systems. Moreover, once the keys for the broadcaster are established, they may be used indefinitely. This process is similar to a "bootstrapping" setup found in related work [4, 29, 41, 58, 90, 95]. Spectrum is agnostic to how this setup takes place: one possibility is to use Riposte [27, 30], which shares a similar threat model.

Step 1: Sharing a message. As in the DC-net scheme, the broadcaster generates secret-shares of the broadcast message $\widehat{m}$ in the field $\mathbb{F}$. All other clients (subscribers) generate secret-shares of the message 0. The only difference is that in Spectrum, the broadcaster knows the broadcast key α while subscribers do not. Let $y = \alpha$, if the client is the broadcaster and $y = 0$ otherwise. Each client proceeds as follows.

1.1: Sample random $m_A, m_B \in \mathbb{F}$ such that $m = m_A + m_B \in \mathbb{F}$.

1.2: Compute $t \leftarrow y \cdot m \in \mathbb{F}$.   // MAC tag (Section 3.1)

1.3: Sample random $t_A, t_B \in \mathbb{F}$ such that $t = t_A + t_B \in \mathbb{F}$.

1.4: Send (m_A, t_A) to ServerA and (m_B, t_B) to ServerB.

The above amounts to secret-sharing the message and access control MAC tag between both servers.

Step 2: Auditing shares. Servers collectively verify access control using the shares of the message and tag.

2.1: ServerA computes $g^{\beta_A} \leftarrow (g^\alpha)^{m_A}/g^{t_A}$.

2.2: ServerB computes $g^{\beta_B} \leftarrow (g^\alpha)^{m_B}/g^{t_B}$.

2.3: The servers swap audit tokens g^{β_A} and g^{β_B} and verify that $g^{\beta_A} \cdot g^{\beta_B} = g^0 = 1_{\mathbb{G}}$.

The above follows the access control verification (Section 3.1). All shares that fail the audit are discarded by both servers. In Section 4, we discuss how we prevent "audit attacks" in which a server tampers with a client request so the check fails.

Step 3: Recovering the broadcast. Servers collectively recover the broadcast message by aggregating all received shares that pass the audit.

3.1: ServerA computes $\mathsf{agg}_A \leftarrow \sum_i (m_A[i]) \in \mathbb{F}$.

3.2: ServerB computes $\mathsf{agg}_B \leftarrow \sum_i (m_B[i]) \in \mathbb{F}$.

3.3: Servers swap agg_A and agg_B.

3.4: Servers compute $\widehat{m} \leftarrow \mathsf{agg}_A + \mathsf{agg}_B \in \mathbb{F}$.

This recovers the broadcast message as in the vanilla DC-net scheme. The recovered message is then made public to all clients (e.g., via a public bulletin board [7, 25]).

3.3 Towards the full protocol

The single-channel scheme presented in Section 3.2 achieves anonymous broadcast while also preventing broadcast disruption by malicious clients. Two problems remain however. First, while the single-channel scheme is fast and robust against malicious clients, it does not efficiently extend to multiple broadcasters. Second, a malicious server can tamper with the audit to make it fail for one or more clients—and learn whether one of them was a broadcaster (see Appendix A).

Supporting multiple channels. To support multiple channels, we use distributed point functions (DPFs) [15, 16, 46] to "compress" secret-shares across multiple instances of the DC-net scheme. DPFs avoid the linear bandwidth overhead of repeating DC-nets for each broadcaster and have been successfully used for anonymous broadcast in other systems [2, 30, 41]. However, without access control, the DPFs must expand to a large space to prevent collisions. We show that our construction for single-channel access control extends to the multi-channel setting, where each broadcaster has a key associated with their allocated channel.

Preventing audit attacks. At a high level, our approach is to commit each server to the shares they receive from a client. In the case of an audit failure, each server efficiently proves that it adhered to protocol to blame the client; if it can't, any honest server aborts Spectrum.

4 Many channels and malicious servers

In this section, we extend the single-channel protocol of Section 3.2 to the multi-channel setting. We first show how to use a DPF to support many broadcast channels with little increase in bandwidth overhead (compared to the one-channel setting), an idea introduced in Riposte [30]. We prevent disruption by augmenting DPFs with the anonymous access control technique from Section 3.1. Prior works [13, 16, 30, 34, 41] describe techniques to verify that a DPF is well-formed, but do not allow for access control. Spectrum does both.

4.1 Tool: distributed point functions

A *point function P* is a function that evaluates to a message m on a single input j in its domain $\{1, \ldots, L\}$ and evaluates to zero on all other inputs $i \neq j$ (equivalently, a vector $(0, 0, \ldots, m, \ldots, 0)$). We define a *distributed* point function: a point function encoded and secret-shared among n keys:

Definition 1 (Distributed Point Function (DPF) [30, 46])**.** *Fix integers $L, n \geq 2$, a security parameter λ, and a message space $\mathcal{M}$. Let $\boldsymbol{e}_j \in \{0, 1\}^L$ be the jth row of the $L \times L$ identity matrix. An n-DPF consists of (randomized) algorithms:*

- $\mathsf{Gen}(1^\lambda, m \in \mathcal{M}, j \in \{1, \ldots, L\}) \rightarrow (k_1, \ldots, k_n)$,
- $\mathsf{Eval}(k_i) \rightarrow (m_1, m_2, \ldots, m_L)$.

These algorithms must satisfy the following properties:

- Correctness. *A DPF is* correct *if expanding the output of* Gen *into the space of L messages $\mathcal{M}^L$ and combining gives the corresponding point function:*

$$\Pr\left[\begin{array}{l}(k_1, \ldots, k_n) \leftarrow \mathsf{Gen}(1^\lambda, m, j) \\ s.t. \ \sum_{i=1}^n \mathsf{Eval}(k_i) = m \cdot \boldsymbol{e}_j\end{array}\right] = 1,$$

where the probability is over the randomness of Gen

- Privacy. *A DPF is* private *if any subset of evaluation keys reveals nothing about the inputs. That is, there exists an efficient simulator* Sim *which generates output computationally indistinguishable from strict subsets of the keys output by* Gen.

We use a DPF with domain $\{1, \ldots, L\}$, where each broadcaster/channel has an index $j \in \{1, \ldots, L\}$. Each broadcaster must write a message m to channel j, but not elsewhere: we can think of this as a point function P with $P(j) = m$. Then, we can encode secret-shares of P using a DPF more efficiently than secret-sharing its vector representation (as in repeated DC-nets). Evaluated DPF shares can still be aggregated locally, and our access control protocol supports DPFs with a slight modification (Section 4.2).

DPFs are concretely efficient. The key size for state-of-the-art 2-DPFs [16] is $O(\log L + |m|)$ (assuming PRGs); for the general case [15], when $n > 2$, the key size is $O(\sqrt{L} + |m|)$ under the decisional Diffie-Hellman assumption [10].
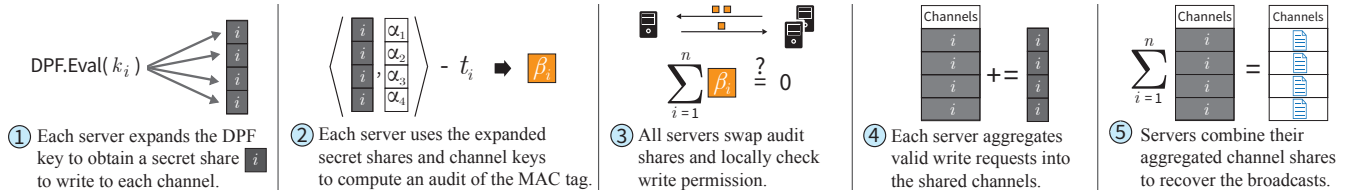
Figure 2: Overview of the server-side pipeline when processing a client's request. Steps ①, ④ and ⑤ are computed over the field $\mathbb{F}$. Steps ② and ③ are computed "in the exponent" of the group $\mathbb{G}$ when using the technique described in "Preventing client-server collusion" of Section 3.1.

Server-side work to expand each DPF uses fast symmetric-key operations in the two-server case [15, 16] and group operations in the multi-server case [30]. With $L = 2^{20}$, the DPF key size for the two-server construction is 325 B and for the $n > 2$ construction 64 kB (excluding the message size).

4.2 Spectrum with many channels

In this section, we present the full Spectrum protocol with L channels and $n \geq 2$ servers. Broadcasters reserve a channel in the setup phase. Clients encode their message at their channel (if any) using a DPF; the servers anonymously audit access to all channels before recovering messages.

Setup. The setup in this setting is similar to the setup described in Section 3.2. Each broadcaster anonymously provides a public verification key g^{α_i} to the servers, to be associated with a channel. In addition to their key, any user with content to broadcast might upload a brief description or "teaser" of their content; the servers can choose which to publish, or users could perform a privacy-preserving vote [28]. We leave detailed exploration of the fair allocation of broadcast slots to users to future work. Post-setup, all servers hold a vector of L verification keys $(g^{\alpha_1}, \ldots, g^{\alpha_L})$. Each key corresponds to one channel.

Step 1: Sharing a message. Let $y = \alpha_j$ and $j' = j$ if the client is a broadcaster for the jth channel ($y = 0$ and $j' = 0$ otherwise). Only broadcasters have $m \neq 0$. Each client runs:

1.1: $(k_1, \ldots, k_n) \leftarrow \mathsf{DPF.Gen}(1^\lambda, m, j')$.   // gen DPF keys
1.2: Compute $t \leftarrow m \cdot y \in \mathbb{F}$.
1.3: Sample $(t_1, \ldots, t_n) \xleftarrow{R} \mathbb{F}$ such that $\sum_{i=1}^n t_i = t \in \mathbb{F}$.
1.4: Send share (k_i, t_i) to the ith server, for $i \in \{1, \ldots, n\}$.

Step 2: Auditing shares. Upon receiving a request share (k_i, t_i) from a client, each server computes:

2.1: $m_i \leftarrow \mathsf{DPF.Eval}(k_i) \in \mathbb{F}^L$.
2.2: $A \leftarrow \prod_{j=1}^L (g^{\alpha_j})^{m_i[j]}$.   // $A = g^{\langle m_i, (\alpha_1, \ldots, \alpha_L) \rangle}$
2.3: $g^{\beta_i} \leftarrow A / g^{t_i}$.
2.4: Send g^{β_i} to all other servers.

All servers check that $\prod_i^n g^{\beta_i} = g^0 = 1_{\mathbb{G}}$. If this condition does not hold, then the client's request is dropped by all servers. In Section 4.3, we show how to detect a malicious server that tampers with a client's request so that it fails this audit.

Step 3: Recovering the broadcast. Each server keeps an accumulator m_i of L entries (i.e., the channels), initialized to $\mathbf{0} \in \mathbb{F}^L$. Let $S = \{(k_j, t_j) \mid j \leq N\}$ be the set of all valid requests that pass the audit of Step 2. Each server:

3.1: Computes $m_i \leftarrow \sum_{(k,t) \in S} \mathsf{DPF.Eval}(k) \in \mathbb{F}^L$.
3.2: Publicly reveals m_i.   // shares of the aggregate.

Using the publicly revealed shares, anyone can recover the L broadcast messages as $\widehat{m} = \sum_i^n m_i \in \mathbb{F}^L$.

4.3 BlameGame: preventing audit attacks

BlameGame is a network overlay protocol that verifies who received what during protocol execution.

We use a *verifiable* encryption scheme [11, 20] where a party with a secret key can prove that a ciphertext decrypts to a certain message (DecProof makes a proof, and VerProof verifies it; see definition in Appendix C.1). Verifiable encryption reveals the plaintext request shares of a client to all servers if the client or the server is malicious (a malicious server may do this once, but will be immediately eliminated). BlameGame also uses a Byzantine broadcast protocol [19] so that all servers get the (encrypted) shares of all other servers.

BlameGame. BlameGame commits clients and servers to specific requests used in the audit. If the audit fails, honest servers reveal (with a publicly verifiable proof) the share they were given, which allows other servers to verify the results of the audit locally, which indicts the client. Dishonest servers cannot give valid proofs for their shares.

Setup. All servers make a key pair $(\mathsf{pk}_i, \mathsf{sk}_i)$ and publish pk_i.

Step 1: Generating commitments. Let τ_i be the client's request secret-share for server i. The client runs:

1.1: $C_i \leftarrow \mathsf{Enc}(\mathsf{pk}_i, \tau_i)$.   // Encryption under pk_i.
1.2: Byzantine broadcast *all* C_i to *all* servers.

Server i recovers $\tau_i \leftarrow \mathsf{Dec}(\mathsf{sk}_i, C_i)$; clients may go offline at this point. All servers are committed to the *encryption* of their secret-shares. We describe an optimization in Section 5.1 that makes the size of each C_i constant.

Step 2: Proving innocence. Each server publishes their share of the request τ_i and a proof of correct decryption:

2.1: $(\pi_i, \tau_i) \leftarrow \mathsf{DecProof}(\mathsf{sk}_i, C_i)$.
2.2: Send (π_i, τ_i) to all servers.

Step 3: Assigning blame. Using the posted shares and proofs, each server assigns blame:

3.1: Collect (π_i, τ_i) from servers $i \in \{1, \ldots, n\}$ and all C_i.

3.2: Check that $\mathsf{VerProof}(\mathsf{pk}_i, \pi_i, C_i, \tau_i) = \mathsf{yes}$, for $1 \le i \le n$.

3.3: Check the audit using all the shares $(\tau_1, \ldots, \tau_n)$.

3.4: Assign blame:
   if 3.2 fails for any i: abort;   `// bad server`
   else if 3.3 passes: abort;   `// bad server`
   else if 3.3 fails: drop the client request. `// bad client`

5   Optimizations and extensions

Here, we describe extensions and optimizations to Spectrum. We show how to (1) broadcast *large* messages efficiently and (2) *privately* fetch published broadcasts as a subscriber.

5.1   Handling large messages efficiently

We described Spectrum in Section 4.2 with messages as elements of a field $\mathbb{F}$, which we check to perform access control. While a 16 B field suffices for audit security, large messages require much larger fields (or repeating the protocol many times). These approaches require proportionally greater bandwidth and computation to audit. Instead, we give a black-box transformation from a 2-server DPF over $\mathbb{F}$ to a DPF over ℓ-bit strings, *preserving security* (see Section 6.2). We use a pseudorandom generator (PRG). Clients create DPF keys encoding a short PRG seed, rather than a message. The servers efficiently audit this seed as before to enforce access control. Then, they expand it to a much longer message with the guarantee that the DPF is still non-zero at an index for which the client knows the broadcast key (if the message is non-zero).

The transformation. Let DPF be a DPF over the field $\mathbb{F}$ and let $\mathsf{DPF}^{\mathsf{bit}}$ be a DPF over $\{0,1\}$. Let $G : \mathbb{F} \to \{0,1\}^\ell$ be a PRG. To write to channel j, a user computes:

1. $\bar{s} \xleftarrow{R} \mathbb{F}$.   `// random nonzero PRG seed`
2. $(k_A, k_B) \leftarrow \mathsf{DPF.Gen}(\bar{s}, j)$.
3. $s_A^* \leftarrow \mathsf{DPF.Eval}(k_A)[j], s_B^* \leftarrow \mathsf{DPF.Eval}(k_B)[j]$.
4. $\bar{m} \leftarrow G(s_A^*) \oplus G(s_B^*) \oplus m$.
5. $(k_A^{\mathsf{bit}}, k_B^{\mathsf{bit}}) \leftarrow \mathsf{DPF}^{\mathsf{bit}}.\mathsf{Gen}(1, j)$.
6. Send $(\bar{m}, k_A, k_A^{\mathsf{bit}})$ to ServerA, $(\bar{m}, k_B, k_B^{\mathsf{bit}})$ to ServerB.

Every server evaluates the DPF keys to a vector $\boldsymbol{s}$, of PRG seeds, and a vector $\boldsymbol{b}$ of bits. Each seed and bit other than the jth is *identical* on both servers (a secret-share of zero); at j, we get $s_A^* \ne s_B^*$. Servers evaluate the DPF by expanding each $\boldsymbol{s}[i]$ to an ℓ-bit string and XORing $\bar{m}$ only when $\boldsymbol{b}[j] = 1$. If we define multiplication of a binary string by a bit as $1 \cdot \bar{m} = \bar{m}$ and $0 \cdot \bar{m} = 0$, ServerA computes:

$$\boldsymbol{m}_A := (G(\boldsymbol{s}_A[1]) \oplus \boldsymbol{b}_A[1] \cdot \bar{m}, \ldots, G(\boldsymbol{s}_A[L]) \oplus \boldsymbol{b}_A[L] \cdot \bar{m}).$$

ServerB does the same. Then, we get that:

$$\boldsymbol{m}_A[i] \oplus \boldsymbol{m}_B[i] = \begin{cases} G(\boldsymbol{s}[i]) \oplus G(\boldsymbol{s}[i]) = 0^\ell & i \ne j \\ G(s_A^*) \oplus G(s_B^*) \oplus \bar{m} = m & i = j. \end{cases}$$

Servers perform the audit (in $\mathbb{F}$) over the expanded PRG seeds and bits as in Section 3.2. Observe that the final output is non-zero only if: (1) some PRG seed, (2) some bit, or (3) the masked message $\bar{m}$ is different on each server. The $\boldsymbol{s}$ and $\boldsymbol{b}$ audit checks (1) and (2); servers check (3) by comparing hashes of $\bar{m}$. As before, the 0 MAC tag passes the audit for an empty message, and broadcasters can provide a correct tag.

Many servers. The above transformation generalizes to the n-server setting. The intuition is the same: only "non-zero" PRG seeds should expand to write non-zero messages. However, we need a PRG with special properties for this to hold with $n > 2$. We give the full transformation in Appendix B. Applying this transformation to a square-root DPF yields the n-server DPF of Corrigan-Gibbs et al. [30], but now with access control.

BlameGame optimization. The masked message $\bar{m}$, given to all servers, constitutes the bulk of data in *each* DPF key, so clients can omit it in their request commitments (Section 4.3) when using the above transformation because servers do not need it to verify access control. (The verification performed by the servers only depends on the DPF *seeds* and checking equality of the masked message $\bar{m}$.)

5.2   Private broadcast downloads

Content published using an anonymous broadcast system is likely to be sensitive and subscribers might want to have plausible deniability when it comes to which broadcasts they are interested in. In a setting with many channels, we might allow the subscribers to download one channel while hiding *which* channel they download: the exact setting of private information retrieval (PIR) [24]. In (multi-server) PIR, a client submits *queries* to two or more servers, receiving *responses* which they combine to recover one document in a "database." The queries hide which document was requested. In Spectrum, clients can use any PIR protocol to hide which channel they download. Modern PIR schemes based on DPFs have minimal bandwidth overhead for queries [15, 16]. However, the processing time on each server is always linear [24]. We evaluate the overhead of using PIR for subscriber anonymity in Section 7.1.

6   Security and efficiency analysis

In this section, we analyze the theoretical efficiency and security of Spectrum with respect to the threat model and required guarantees outlined in Section 2.4.

6.1 Efficiency analysis

We briefly analyze the efficiency of Spectrum (Section 4.2) and BlameGame (Section 4.3) with the above optimizations.

Communication efficiency in Spectrum. Spectrum can use any DPF construction with outputs in a finite field using the transformation of Section 5.1 to support ℓ-bit messages with only an additive $O(\ell)$ overhead to the DPF key size. Using optimized two-server DPF constructions [15, 16], clients send requests of size $O(\log L + |m|)$ (for L channels). With more than two servers, the communication is $O\left(\sqrt{L} + |m|\right)$ using the seed-homomorphic PRG based DPF construction [30]. For the audit, inter-server communication is constant.

Computational efficiency in Spectrum. Each server performs $O(L \cdot |m|)$ work per client when aggregating the shares and performing the audit ($O(N \cdot L \cdot |m|)$ total for N clients). The work on each client is $O(\log L + |m|)$ when using two-server DPFs and $O(\sqrt{L} + |m|)$ otherwise [15].

6.2 Security of Spectrum

We first describe the ideal functionality of the anonymous broadcast system which Spectrum instantiates.

Ideal functionality. Ideal Spectrum is defined as follows:

- Receive message $m = 0$ from each subscriber, $m = \widehat{m}$ from the broadcaster, and no input from the servers.
- Output $\widehat{m}$ to both the clients and servers.

Client anonymity. We argue that Spectrum provides client anonymity by constructing a simulator for the view of a network adversary corrupting any strict subset of servers.

Claim 1. *If at least one server is honest, then no probabilistic polynomial time (PPT) adversary $\mathcal{A}$ observing the entire network and corrupting any strict subset of the servers and an arbitrary subset of clients, can distinguish between an honest broadcaster and an honest subscriber.*

Proof. We construct a simulator Sim for the view of $\mathcal{A}$ when interacting with an honest client. Let $\widehat{\text{Sim}}$ be the DPF simulator (see Definition 1). Sim proceeds as follows:

1. Take as input $(\mathbb{G}, g)$, $(g^{\alpha_1}, \ldots, g^{\alpha_L})$, $\mathbb{F}$, and subset of corrupted server indices $I \subset \{1, \ldots, n\}$.
2. Sample $t_i \xleftarrow{R} \mathbb{F}$ for $i \in \{1, \ldots, n\}$ such that $\sum_i t_i = 0$.
3. $\{k_i \mid i \in I\} \leftarrow \widehat{\text{Sim}}(I)$.   // see Definition 1
4. Output $\text{View} = \left(\{(t'_i, k_i) \mid i \in I\}, \{g^{t_j} \mid j \in \{1, \ldots, n\} \setminus I\}\right)$.

Analysis. The view includes:

- Each DPF key k_i for corrupted server i.
- Each MAC tag share t'_i for corrupted server i.
- Audit shares g^{t_j} from every honest server j.

The DPF keys are computationally indistinguishable from real DPF keys by the security of the DPF simulator. Therefore, it remains to argue that the tag and audit shares are distributed identically to the real view. Recall that during an audit, server i publishes $g^{\beta_i} = g^{\langle m_i, (\alpha_1, \ldots, \alpha_L) \rangle - t_i}$ where m_i is the output of DPF.Eval(k_i) and t_i is a secret-share of the MAC tag t. For a subscriber, $\langle m_i, (\alpha_1, \ldots, \alpha_L) \rangle$ (the inner product) gives a random secret share of 0 and t_i is a secret share of 0, so g^{β_i} is a random (multiplicative) secret share of g^0. For a broadcaster publishing to channel j, $\langle m_i, (\alpha_1, \ldots, \alpha_L) \rangle$ is a random secret share of $m \cdot \alpha_j = t$, so g^{β_i} as computed by the ith server is a random multiplicative secret share of g^0 as well. Therefore, the distribution of the audit and tag shares (g^{β_i} and t_i, respectively) is identical to the real view. Finally, because the connection between clients and servers is encrypted (and of fixed-size), we can efficiently simulate network traffic as random encrypted data. □

Disruption resistance in Spectrum. We prove that a client cannot disrupt a broadcast on the jth channel without knowing the channel broadcast key α_j.

Claim 2. *Assuming the hardness of the discrete logarithm problem [11, 40] in $\mathbb{G}$, no probabilistic polynomial time (PPT) client can write to channel j and pass the audit performed by the servers without knowledge of α_j.*

Proof. Assume towards contradiction that some adversarial client can generate (potentially ill-formed) DPF keys that result in a non-zero vector (WLOG, assume that index L is non-zero) and pass the audit for a given access tag with non-negligible probability. We can use the client to extract the discrete logarithm for any element of $\mathbb{G}$ as follows. Given g^{α^*}, choose random $\alpha_i \in \mathbb{F}$ for $i \in \{1, \ldots, L-1\}$. Give the client $\left(g^{\alpha_1}, \ldots, g^{\alpha_{L-1}}, g^{\alpha^*}\right)$ and get in return DPF keys $(k_1, \ldots k_n)$ and MAC tag t. Given these DPF keys, we can compute $m = (m_1, \ldots, m_L)$ by evaluating the DPF. If the shares pass the audit, it must be that $\langle m, \alpha \rangle = t$. However, α includes α^* so we can solve for α^* (t and all α_i except for α^* are known). We conclude that the client has knowledge of α^*. □

Security of the DPF transformation. The construction from Section 5.1 maintains security. This construction transforms a DPF DPF into a DPF DPF′ over ℓ-bit messages.

Claim 3. *If Spectrum with DPF preserves client anonymity, Spectrum with DPF′ preserves client anonymity.*

Proof. We build a simulator Sim′ for DPF′ from the simulator Sim for DPF. Sim′ simply runs Sim twice (once to generate the seed-DPF keys and once for the bit-DPF keys) and picks an ℓ-bit message uniformly at random for $\bar{m}$. The simulator's $\bar{m}$ is computationally indistinguishable from the real $\bar{m}$ (otherwise, the PRG used to mask the message is not secure). Therefore, if there exists an efficient distinguisher, it can also

distinguish between the keys output by Sim and the real DPF keys, a contradiction. □

Claim 4. *If Spectrum with* DPF *has disruption resistance, Spectrum with* DPF′ *has disruption resistance.*

Proof. Assume, towards contradiction, that there exists a computationally bounded adversary $\mathcal{A}$ which does *not* obtain the broadcast key α as input, and outputs a set of DPF′ keys along with MAC tag shares. If the set of DPF′ keys write to at least one channel and the tag shares output by $\mathcal{A}$ pass the server MAC audit, then we can produce a non-zero message and tag for DPF as follows. WLOG, we fix the number of servers to $n = 2$. Run $\mathcal{A}$ to get two DPF′ keys k_1', k_2' and tag $t = (t_1, t_2)$. By construction, $k_i' = (k_i, k_i^{\mathrm{bit}}, m')$, where k_i and k_i^{bit} are DPF keys with range $\mathbb{F}_p$ and $\mathbb{F}_2$, respectively, and $m' \in \mathbb{F}_{2^\ell}$ is a masked message (identical in each DPF′ key). If these keys and tags pass the audit, the masked message in each key is the same (by the collision resistance of the audit hash function). Then, because the key for DPF′ writes a non-zero message, at least one of the two DPF keys (either k or k^{bit}) must write a non-zero message (otherwise the keys would be writing zero). If follows that (k_1, k_2) or $(k_1^{\mathrm{bit}}, k_2^{\mathrm{bit}})$ encode a non-zero message, which contradicts Claim 2. □

6.3 Security of BlameGame

We must show that in BlameGame: (1) an honest client will never be blamed, (2) a malicious client will always be blamed, (3) an honest server will never be blamed, and (4) a malicious server will always be blamed. Incorrect blame attribution indicates a failure of the verifiable encryption scheme or audit security; see Appendix C.2 for full proof.

Overhead of BlameGame. BlameGame requires some extra bandwidth and computation time. Clients send a shared message mask *once* to each server; DPF keys add about 100 bytes per client request (details in Section 5.1). The servers must run BlameGame for each malicious client. However, verifying decryption takes tens of *microseconds*, and running the audit is similarly quick (see Section 7.1). Because the servers delay the work of aggregating messages until *after* the audit, a malicious client often requires *fewer* cycles than an honest one (but extra network communication).

7 Evaluation

We build and evaluate Spectrum, comparing it to state-of-the-art anonymous broadcasting works: Riposte, Blinder, Express, and Dissent (see related work; Section 8).

Riposte [30] is designed for anonymous broadcasting where all users broadcast at all times. Riposte uses three servers (one trusted for audits) but generalizes to many servers (one honest). Riposte was designed for smaller messages and the source code fails to run with messages of size 5 kB or greater.

Blinder [2] builds on Riposte but requires an *honest majority* of at least 5 servers. Like Riposte, Blinder also assumes that all users are broadcasting. Blinder supports using a server-side GPU to increase throughput.

Express [41] is an anonymous communication system designed for anonymous "dropbox"-like applications. It does not support broadcast as-is, but can be easily modified to do so. We include Express in our comparison as a recent, high-performance system decoupling broadcasters and subscribers.

Dissent [29, 95] has a setup phase (like Spectrum's), a DC-net phase, and a blame protocol. We give measurements both with and without the blame protocol and exclude the setup phase. Without the blame protocol, the system runs a plain DC-net without any disruption resistance and is quite fast. If *any* user sends an invalid message, Dissent runs the (expensive) blame protocol (up to once per malicious user).

We use data from the Blinder paper [2, Fig. 4] as the source did not compile. The Dissent code (last modified in 2014) ran with up to 1000 users and 10 kB messages, but hung indefinitely after increasing either (though the authors report 128 kB messages with 5000 users). Linearly scaling our measurements, we find them broadly consistent (3× faster) with the authors' reported measurements for 128 kB messages with the same number of users in a similar setting [95, Fig. 7].

Implementation. We build Spectrum in ~8000 lines of open-source [1] Rust code, using AES-128 (CTR) as a PRG and BLAKE3 [66] as a hash. Because our DPF has relatively few "channels" L, a DPF with $O(L)$-sized keys (adapted from Corrigan-Gibbs et al. [30]) gives the best concrete performance. For the multi-server extension (Section 5.1 and Appendix B), we use a seed-homomorphic PRG [12] with the Jubjub [49] curve. We encrypt traffic with TLS 1.3 [73].

Environment. We run VMs on Amazon EC2 to simulate a WAN deployment. Each `c5.4xlarge` 8-core instance has 32 GiB RAM [76], running Ubuntu 20.04 ($0.68 per hour in September 2021). We run clients in `us-east-2` (Ohio) and servers in `us-east-1` (Virginia) and `us-west-1` (California). Network round trip times (RTTs) were 11 ms between Virginia and Ohio, 50 ms between Ohio and California, and 61 ms between Virginia and California. Inter-region bandwidth was 524 Mbit/s (shared between many clients simulated on the same machine).

7.1 Results

In our experiments, we find Spectrum is 4–7× faster than Express for 5 MB to 100 kB messages, 2× / 13–17× slower than Blinder (CPU/GPU, resp.) in *unfavorable* settings, 500–7500× / 250–520× faster than Blinder (CPU/GPU) in *favorable* settings, and 16–12,500× faster than Riposte. We run 5 trials per setting, shading the 95% confidence interval (occasionally too small to be seen).
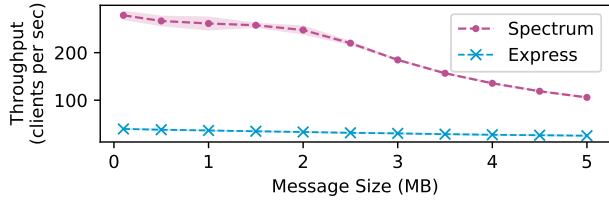
Figure 3: Throughput (client requests per second; higher is better) for a one channel deployment (one broadcaster and many subscribers).
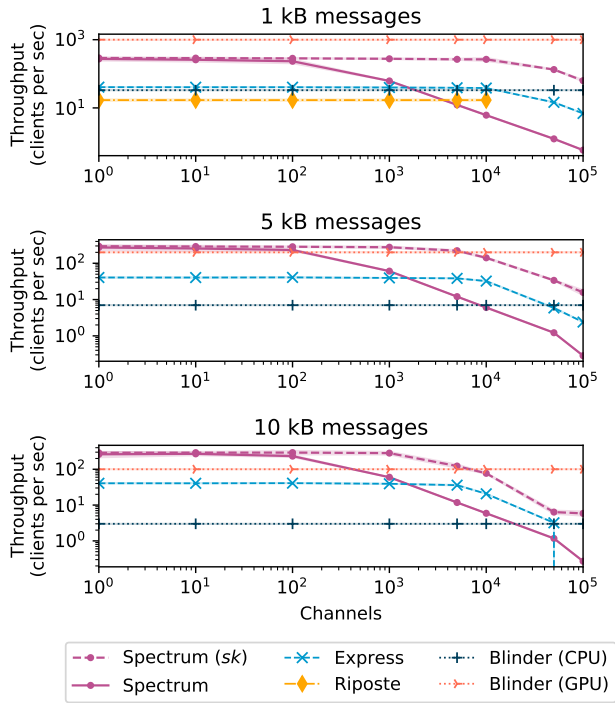
Figure 4: Throughput (requests per second; higher is better) for broadcasts with 100,000 users with varying numbers of broadcasting users ("channels"): Express and Spectrum benefit from fewer channels. (Blinder numbers as reported by the authors [2].)

One channel. In Figure 3, we report the throughput (client requests per second) for both Spectrum and Express in the one-channel setting. As expected, performance is worse with larger messages for both systems. However, we find that Spectrum, compared to Express, is 4–7× faster on messages between 100 kB and 5 MB. Riposte and Blinder have no analog for the single-channel setting. (Dissent *does* support a one-channel setting, but did not run with large messages.)

Many channels. Unlike Riposte and Blinder, Spectrum is faster with fewer broadcasters. To compare, we fix 100,000 users and vary the number of channels from 1 (best-case for Spectrum) to 100,000 (worst-case). We evaluate Spectrum with and without the change described in "Preventing client-server collusion" (Section 3.1). Without the change, which we call *"Spectrum (sk)"*, servers obtain the MAC secret key for each channel. This mirrors the threat model of e.g., Ex-

Request Size	**Request** per client	**Audit** per client	**Aggregation** once per server
	$|m|$+ 70 bytes	70 bytes	$|m|$+ 3 bytes
BlameGame (per failed audit)	**Backup Request** per client	**Audit** per client	**Decryption** once per client
	140 bytes	200 bytes	10 µs

Table 2: Upper bound on request size for one channel and $|m|$-bit messages. BlameGame only runs if the first request audit fails.
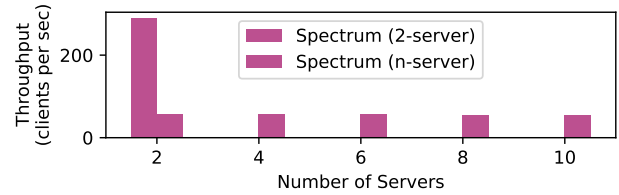
Figure 5: Spectrum can generalize to $n > 2$ servers (shown for 10 kB messages). This uses an expensive PRG and is therefore slower, but adding more servers causes no additional slowdown.

press [41]. *With the change*, servers only get MAC *public keys* which prevents covert client-server collusion. However, there is a modest price in terms of performance due to the elliptic curve operations (see Figure 4). We find that Spectrum (both variants) outperform all other systems with 10 kB messages for relatively few channels (up to hundreds), but performs relatively worse with smaller messages or more channels. For "Twitter-like" settings, another system (e.g., Blinder or Riposte) may be appropriate.

Overhead. In any anonymous broadcast scheme, every client (even subscribers) must upload data corresponding to the message length $|m|$ to ensure privacy. For DC-net based schemes, the client sends a size-$|m|$ request to each server. We measure the concrete request sizes of Spectrum and compare to this baseline in Table 2. Client request overhead is small: about 70 B, roughly 75× smaller than in Express. Moreover, in Spectrum, request audits are under 16 B, a 120× improvement over Express [41]. BlameGame imposes little overhead (both in terms of bandwidth and computation). Because BlameGame runs only when a request audit fails, these overheads occur for few requests in most settings.

Many servers. In Section 5.1 and Appendix B, we note that our construction of Spectrum generalizes from 2 to n servers (with one honest) in a manner similar to Riposte [30]. The n-server construction uses a seed-homomorphic pseudorandom generator (PRG) [12]. On one core of an AMD Ryzen 4650G CPU, we measured the maximum throughput of our seed-homomorphic PRG at 300 kB/s, 20,000 times slower than an AES-based PRG. For 10,000 kB messages, Spectrum was 5× slower with the seed-homomorphic PRG (Figure 5); with
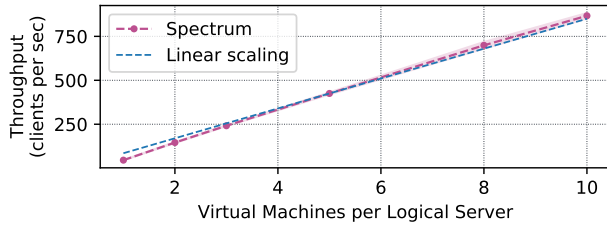
Figure 6: Spectrum is highly parallelizable: for 500 channels of 100 kB messages, 10 VMs per "server" gives a 10× speedup.

Figure 7: Latency for uploading a single document with varying numbers of users. Blinder numbers as reported by the authors [2, Fig. 4] and linearly scaled to 1 MB messages.

L	up	down
10^1	1.25 B	$|m|$
10^2	12.5 B	$|m|$
10^3	125 B	$|m|$
10^4	1.25 kB	$|m|$
10^5	12.5 kB	$|m|$

Figure 8: **Left:** Bandwidth usage of a PIR query with varying number of channels L. **Right:** Server capacity (one core) to answer PIR queries for private client downloads. For L channels, the client requests one out of L documents, where channels have size $|m|$.

larger messages, the relative difference increases. We find *no additional* slowdown between 2 to 10 servers. An interesting direction for future work would be to evaluate Spectrum with LWE-based seed-homomorphic PRG constructions [12], as they are likely to have better concrete performance.

Scalability. We may trust machines administered by the same *organization* equally, viewing several worker servers as one logical server. Client requests trivially parallelize across such workers: running 10 workers per logical server leads to a 10× increase in overall throughput (Figure 6). In a cloud deployment, Spectrum handles the same workload in less time for negligible additional cost by parallelizing the servers.

Latency. In Figure 7, we measure the time to broadcast a single document for these systems with varying numbers of users. For Spectrum, we use a 1 MB message. For Blinder, we use numbers reported by the authors [2, Fig. 4], multiplied to the same message size (the authors explicitly state that repeating the scheme many times is the most efficient way to send large messages). We benchmark Dissent both with and without the blame protocol invoked during a round. The former (blame) is the performance of Dissent if any client misbehaves. The latter (no blame) assumes that no client misbehaves. Express doesn't have a notion of "rounds" so we omit it here. We find that for one channel of large messages, Spectrum is much faster than other systems (except Dissent with no blame protocol; i.e., when all clients are honest).

Client privacy. In Section 5.2, we outlined how private information retrieval (PIR) [24] techniques provide client privacy for multiple channels. Figure 8 shows the server-side CPU capacity to process these requests for 1 kB, 10 kB, and 100 kB messages and 1–100,000 channels. We measure one core of an AMD Ryzen 4650G CPU for a simple 2-server PIR construction [24], finding good concrete performance.

7.2 Discussion

Our evaluations showcase the use of Spectrum for a real-world anonymous broadcasting deployment using commodity servers. Compared to the state-of-the-art in anonymous broadcasting, Spectrum achieves speedups in settings with a large ratio of passive subscribers to broadcasters. Based on our evaluation, with 10,000 users, Spectrum could publish: a
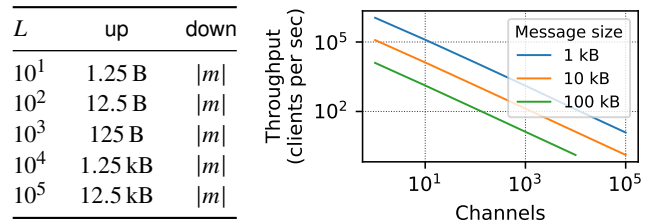
PDF document (1 MB) in 50s, a *podcast* (50 MB) in 40m, or a *documentary movie* (500 MB, the size of Alexei Navalny's documentary on Putin's Palace at 720p [80]) in 6h40m.

Operational costs. We estimate costs for a cloud deployment of Spectrum using current Amazon EC2 prices, reported in US dollars. Servers upload about 100 bytes per query (in the above settings, at most 1 GB per day) and inbound traffic is free on EC2. We focus on compute costs: $6.84 per GB published through Spectrum (with 10,000 users). Table 3 compares costs to publish 1 GB among 10,000 users.

8 Related work

Existing systems for anonymous broadcast are suitable for 140 B to 40 kB [2, 30, 41] broadcasts, orders of magnitude smaller than large data dumps [69, 75, 77] common today.

Mix Networks and Onion Routing. In a mix net [22], users send an encrypted message to a proxy server, which collects and forwards these messages to their destinations in a random order. Chaining several such hops protects users from compromised proxy servers and a passive network adversary. Mix

System	Cost (USD)
Blinder (GPU)	$2,000,000.00
Blinder (CPU)	$250,000.00
Riposte	$218,000.00
Dissent (with blame protocol; one round)*	$76,000.00
Dissent (honest clients)	$134.00
Express	$30.22
Spectrum	$6.84

Table 3: Cost to upload one 1 GB document anonymously with 10,000 users, based on the *best* observed rate for each system with that many users (that is, the maximum throughput over all settings we measured; for Blinder, we use the best reported rate). We multiply the total time at the maximum throughput by hourly rate to get the cost. *Extrapolated from 1000 users.

nets and their variations [32, 56, 57, 59, 60, 63, 64, 71, 72, 81, 82, 90] scale to many servers. However, because messages are exchanged and shuffled between many servers, mix nets are poorly suited to high-bandwidth applications. Atom [55] uses mix nets with zero-knowledge proofs to horizontally scale anonymous broadcast to millions of users (Spectrum achieves about 12,500× the throughput [55, Fig. 9]). Riffle [54] uses a *hybrid verifiable shuffle*; in the broadcast setting, it shares a 300 MB file with 500 users in 3 hours (Spectrum supports about 10,000 users in that time).

Some systems use onion routing for better performance than a mix net. In onion routing, users encrypt their messages several times (in onion-like layers) and send them to a chain of servers. Tor [37], the most popular onion routing system, has millions of daily users [83]. Tor provides security in many real-world settings, but is vulnerable to traffic analysis [53, 62, 78]. If only one user sends large volumes of data, an adversary can identify them—Tor discourages high bandwidth applications for this and other reasons [36].

DC-nets. Another group of anonymous communications systems use dining cryptographer networks (DC-nets) [23] (Section 2). DC-nets are vulnerable to disruption: any malicious participant can clobber a broadcast by sending a "bad" share. Dissent [29, 95] augments the DC-nets technique with a system for accountability. Like Spectrum, Dissent performs best if relatively few users are broadcasting. The core data sharing protocol is a standard DC-net, which is very fast and supports larger messages. Further, it supports many servers at little additional cost. However, Dissent is not suitable for many-user applications where disruption is a concern. If *any* user misbehaves, Dissent must undergo an expensive blame protocol (quadratic in the total number of users). This approach detects, rather than prevents, disruption. The user is evicted after this protocol, but an adversary controlling many users can cause many iterations of the blame protocol.

PriFi [6] builds on the techniques in Dissent to create indistinguishability among clients in a LAN. Outside servers

help disguise traffic using low-latency, precomputed DC-nets. Like Dissent, PriFi catches disruption after-the-fact using a blame protocol (as often as once per malicious user). The PriFi blame algorithm is much faster, but still scales with *all* users in the system (in Spectrum, each malicious user incurs constant server-side work).

Riposte [30] enables anonymous Twitter-style broadcast with many users using a DC-net based on DPFs and an auditing server to prevent disruptors. We find that Riposte is 16× slower than Spectrum with 10,000 users. Further, Riposte assumes that all users are broadcasting and therefore gets *quadratically* slower in the total number of users.

A more recent work, Blinder [2] uses multi-party computation to prevent disruption. Blinder's threat model requires at least five servers with an honest majority. Like Spectrum, Blinder is resilient to active attacks by a malicious server. It is fast for small messages when most users have messages to share, but much slower for large messages. Blinder allows trading money for speed with a GPU.

Express [41] is a system for "mailbox" anonymous communication (writing anonymously to a designated mailbox). Express also uses DPFs for efficient write requests. However, it only runs in a two-server deployment. Express is *not* a broadcasting system, and while it is possible to adapt it to work in a broadcast setting, it is not designed to withstand active attacks by the servers and is insecure for such an application (see Appendix A for details).

9   Conclusions

Spectrum supports high-bandwidth, low-latency broadcasts from a small set of broadcasters to a large number of subscribers by applying new tools to the classic DC-net architecture. We prevent disruption by malicious clients with an efficient blind access control mechanism that prevents clients from writing to a channel they do not have access to.

Additionally, we introduce optimizations to decouple server-side overhead from the message size, which allows Spectrum to scale to large messages and many broadcasters. To prevent malicious servers from deanonymizing clients, we develop a lightweight blame protocol to abort Spectrum if a server deviates from the protocol. Our experimental results show that Spectrum can be used for uploading gigabyte-sized documents anonymously among 10,000 users in 14 hours.

10   Acknowledgments

References

[1] Spectrum implementation. https://www.github.com/znewman01/spectrum-impl, 2021.

[2] Ittai Abraham, Benny Pinkas, and Avishay Yanai. Blinder: Scalable, robust anonymous committed broadcast. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, pages 1233–1252, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370899. doi: 10.1145/3372297.3417261. URL https://doi.org/10.1145/3372297.3417261.

[3] C. Fred Alford. Whistleblowers and the narrative of ethics. *Journal of social philosophy*, 32(3):402–418, 2001.

[4] Sebastian Angel and Srinath Setty. Unobservable communication over fully untrusted infrastructure. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 551–569, 2016.

[5] Raymond Walter Apple Jr. 25 years later; lessons from the Pentagon Papers. *The New York Times*, 23 June 1996. URL https://www.nytimes.com/1996/06/23/weekinreview/25-years-later-lessons-from-the-pentagon-papers.html. Accessed March 2022.

[6] Ludovic Barman, Italo Dacosta, Mahdi Zamani, Ennan Zhai, Apostolos Pyrgelis, Bryan Ford, Joan Feigenbaum, and Jean-Pierre Hubaux. Prifi: Low-latency anonymity for organizational networks. *Proc. Priv. Enhancing Technol.*, 2020(4):24–47, 2020. doi: 10.2478/popets-2020-0061. URL https://doi.org/10.2478/popets-2020-0061.

[7] Josh Daniel Cohen Benaloh. *Verifiable secret-ballot elections*. PhD thesis, Yale University, 1987.

[8] Charles Berret. Guide to SecureDrop, 2016. URL https://www.cjr.org/tow_center_reports/guide_to_securedrop.php.

[9] Sanjit Bhat, David Lu, Albert Kwon, and Srinivas Devadas. Var-CNN: A data-efficient website fingerprinting attack based on deep learning. *Proceedings on Privacy Enhancing Technologies*, 2019(4):292–310, 2019.

[10] Dan Boneh. The decision Diffie-Hellman problem. In *Algorithmic Number Theory, Third International Symposium, ANTS-III, Portland, Oregon, USA, June 21-25, 1998, Proceedings*, pages 48–63, 1998. doi: 10.1007/BFb0054851. URL https://doi.org/10.1007/BFb0054851.

[11] Dan Boneh and Victor Shoup. A graduate course in applied cryptography. *Recuperado de https://crypto.stanford. edu/~ dabo/cryptobook/BonehShoup_0_4. pdf*, 2017.

[12] Dan Boneh, Kevin Lewi, Hart Montgomery, and Ananth Raghunathan. Key homomorphic PRFs and their applications. In *Annual Cryptology Conference*, pages 410–428. Springer, 2013.

[13] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Lightweight techniques for private heavy hitters. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 762–776. IEEE, 2021.

[14] Nikita Borisov, George Danezis, Prateek Mittal, and Parisa Tabriz. Denial of service or denial of security? In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 92–102, 2007.

[15] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015*, pages 337–367, Berlin, Heidelberg, 2015. Springer. ISBN 978-3-662-46803-6.

[16] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1292–1303, 2016.

[17] Russ Buettner, Susanne Craig, and Mike McIntire. Long-concealed records show Trump's chronic losses and years of tax avoidance. *The New York Times*, 2020. URL https://www.nytimes.com/interactive/2020/09/27/us/donald-trump-taxes.html. Accessed March 2022.

[18] Bryan Burrough, Sarah Ellison, and Suzanna Andrews. The Snowden saga: A shadowland of secrets and light. *Vanity Fair*, 2014. URL https://www.vanityfair.com/news/politics/2014/05/edward-snowden-politics-interview. Accessed March 2022.

[19] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In Joe Kilian, editor, *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*, volume 2139 of *Lecture Notes in Computer Science*, pages 524–541. Springer, 2001. doi: 10.1007/3-540-44647-8_31. URL https://doi.org/10.1007/3-540-44647-8_31.

[20] Jan Camenisch and Victor Shoup. Practical verifiable encryption and decryption of discrete logarithms. In

Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 126–144. Springer, 2003. doi: 10.1007/978-3-540-45146-4_8. URL https://doi.org/10.1007/978-3-540-45146-4_8.

[21] J Lawrence Carter and Mark N Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979.

[22] David Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–90, 1981.

[23] David Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, 1(1):65–75, 1988.

[24] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*, pages 41–50. IEEE, 1995.

[25] Arka Rai Choudhuri, Matthew Green, Abhishek Jain, Gabriel Kaptchuk, and Ian Miers. Fairness in an unfair world: Fair multiparty computation from public bulletin boards. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 719–728, 2017.

[26] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the Signal messaging protocol. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 451–466. IEEE, 2017.

[27] Henry Corrigan-Gibbs. *Protecting Privacy by Splitting Trust*. PhD thesis, Stanford University, 2019.

[28] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 259–282, 2017.

[29] Henry Corrigan-Gibbs and Bryan Ford. Dissent: Accountable anonymous group messaging. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, pages 340–350. ACM, 2010.

[30] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. Riposte: An anonymous messaging system handling millions of users. In *2015 IEEE Symposium on Security and Privacy*, pages 321–338. IEEE, 2015.

[31] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Annual Cryptology Conference*, pages 643–662. Springer, 2012.

[32] George Danezis, Roger Dingledine, and Nick Mathewson. Mixminion: Design of a Type III anonymous remailer protocol. In *2003 Symposium on Security and Privacy, 2003.*, pages 2–15. IEEE, 2003.

[33] Alex Davidson, Ian Goldberg, Nick Sullivan, George Tankersley, and Filippo Valsorda. Privacy Pass: Bypassing internet challenges anonymously. *Proc. Priv. Enhancing Technol.*, 2018(3):164–180, 2018.

[34] Leo de Castro and Antigoni Polychroniadou. Lightweight, maliciously secure verifiable function secret sharing. *Cryptology ePrint Archive*, 2021.

[35] Candice Delmas. The ethics of government whistleblowing. *Social Theory and Practice*, pages 77–105, 2015.

[36] Roger Dingledine. BitTorrent over Tor isn't a good idea, Apr 2010. URL https://blog.torproject.org/bittorrent-over-tor-isnt-good-idea.

[37] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. Technical report, Naval Research Lab Washington DC, 2004.

[38] Emily Dreyfuss. Chelsea Manning walks back into a world she helped transform, 2017. URL https://www.wired.com/2017/05/chelsea-manning-free-leaks-changed/.

[39] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *Advances in Cryptology - CRYPTO '92, 12th Annual International Cryptology Conference, Santa Barbara, California, USA, August 16-20, 1992, Proceedings*, pages 139–147, 1992. doi: 10.1007/3-540-48071-4_10. URL https://doi.org/10.1007/3-540-48071-4_10.

[40] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[41] Saba Eskandarian, Henry Corrigan-Gibbs, Matei Zaharia, and Dan Boneh. Express: Lowering the cost of metadata-hiding communication with cryptographic privacy. In *30th USENIX Security Symposium (USENIX Security 21)*, Vancouver, B.C., August 2021. USENIX Association. URL https://www.usenix.org/conference/usenixsecurity21/presentation/eskandarian.

[42] Nathan S Evans, Roger Dingledine, and Christian Grothoff. A practical congestion attack on Tor using long paths. In *USENIX Security Symposium*, pages 33–50, 2009.

[43] Cassi Feldman. 60 Minutes' most famous whistleblower. *CBS News*, 2016. URL https://www.theguardian.com/world/2010/nov/28/how-us-embassy-cables-leaked. Accessed March 2022.

[44] Lorenzo Franceschi-Bicchierai. Snowden's favorite chat app is coming to your computer. *Vice*, 2015. URL https://www.vice.com/en/article/signal-snowdens-favorite-chat-app-is-coming-to-your-computer. Accessed March 2022.

[45] Anita Gates and Katharine Q. Seelye. Linda Tripp, key figure in Clinton impeachment, dies. *The New York Times*, 2020. URL https://www.nytimes.com/2020/04/08/us/politics/linda-tripp-dead.html. Accessed March 2022.

[46] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology – EUROCRYPT 2014*, pages 640–658, Berlin, Heidelberg, 2014. Springer. ISBN 978-3-642-55220-5.

[47] Robert D'A Henderson. Operation Vula against apartheid. *International Journal of Intelligence and Counter Intelligence*, 10(4):418–455, 1997.

[48] Nicholas Hopper, Eugene Y Vasserman, and Eric Chan-Tin. How much anonymity does network latency leak? *ACM Transactions on Information and System Security (TISSEC)*, 13(2):1–28, 2010.

[49] Daira Hopwood. Jubjub supporting evidence. https://github.com/daira/jubjub, 2017. Accessed March 2022.

[50] Bastien Inzaurralde. The Cybersecurity 202: Leak charges against Treasury official show encrypted apps only as secure as you make them. *The Washington Post*, 2018.

[51] Markus Jakobsson and Ari Juels. Proofs of work and bread pudding protocols. In *Secure Information Networks: Communications and Multimedia Security, IFIP TC6/TC11 Joint Working Conference on Communications and Multimedia Security (CMS '99), September 20-21, 1999, Leuven, Belgium*, pages 258–272, 1999.

[52] Laurie Kazan-Allen. In memory of Henri Pezerat. http://ibasecretariat.org/mem_henri_pezerat.php, 2009. Accessed March 2022.

[53] Albert Kwon, Mashael AlSabah, David Lazar, Marc Dacier, and Srinivas Devadas. Circuit fingerprinting attacks: Passive deanonymization of Tor hidden services. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 287–302, 2015.

[54] Albert Kwon, David Lazar, Srinivas Devadas, and Bryan Ford. Riffle. *Proceedings on Privacy Enhancing Technologies*, 2016(2):115–134, 2016.

[55] Albert Kwon, Henry Corrigan-Gibbs, Srinivas Devadas, and Bryan Ford. Atom: Horizontally scaling strong anonymity. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 406–422. ACM, 2017.

[56] Albert Kwon, David Lu, and Srinivas Devadas. XRD: Scalable messaging system with cryptographic privacy. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 759–776, Santa Clara, CA, February 2020. USENIX Association. ISBN 978-1-939133-13-7. URL https://www.usenix.org/conference/nsdi20/presentation/kwon.

[57] David Lazar, Yossi Gilad, and Nickolai Zeldovich. Karaoke: Distributed private messaging immune to passive traffic analysis. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 711–725, 2018.

[58] David Lazar, Yossi Gilad, and Nickolai Zeldovich. Yodel: strong metadata security for voice calls. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 211–224, 2019.

[59] Stevens Le Blond, David Choffnes, Wenxuan Zhou, Peter Druschel, Hitesh Ballani, and Paul Francis. Towards efficient traffic-analysis resistant anonymity networks. *ACM SIGCOMM Computer Communication Review*, 43 (4):303–314, 2013.

[60] Stevens Le Blond, David Choffnes, William Caldwell, Peter Druschel, and Nicholas Merritt. Herd: A scalable, traffic analysis resistant anonymity network for VoIP systems. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 639–652, 2015.

[61] Jason Leopold, Anthony Cormier, John Templon, Tom Warren, Jeremy Singer-Vine, Scott Pham, Richard Holmes, Azeen Ghorayshi, Michael Sallah, Tanya Kozyreva, and Emma Loop. The FinCEN Files. *BuzzFeed News*, 2020. URL https://www.buzzfeednews.com/article/jasonleopold/fincen-files-financial-scandal-criminal-networks. Accessed March 2022.

[62] Shuai Li, Huajun Guo, and Nicholas Hopper. Measuring information leakage in website fingerprinting attacks and defenses. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1977–1992, 2018.

[63] Donghang Lu, Thomas Yurek, Samarth Kulshreshtha, Rahul Govind, Aniket Kate, and Andrew Miller. Honey-BadgerMPC and AsynchroMix: Practical asynchronous MPC and its application to anonymous communication. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 887–903, 2019.

[64] Prateek Mittal and Nikita Borisov. ShadowWalker: Peer-to-peer anonymous communication using redundant structured topologies. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 161–172, 2009.

[65] Prateek Mittal, Ahmed Khurshid, Joshua Juen, Matthew Caesar, and Nikita Borisov. Stealthy traffic analysis of low-latency anonymous communication using through-put fingerprinting. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 215–226, 2011.

[66] Jack O'Connor, Samuel Neves, Jean-Philippe Aumasson, and Zooko Wilcox-O'Hearn. BLAKE3: One function, fast everywhere, 2020. URL https://github.com/BLAKE3-team/BLAKE3-specs/blob/master/blake3.pdf. Accessed March 2022.

[67] John O'Connor. "I'm the guy they called Deep Throat". *Vanity Fair*, 2006. URL https://www.vanityfair.com/news/politics/2005/07/deepthroat200507. Accessed March 2022.

[68] Lasse Overlier and Paul Syverson. Locating hidden servers. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 15–114. IEEE, 2006.

[69] Paradise Papers reporting team. Paradise Papers: Tax haven secrets of ultra-rich exposed. *BBC News*, 2017. Accessed March 2022.

[70] D. Phillips. Reality Winner, former NSA translator, gets more than 5 years in leak of Russian hacking report. *The New York Times*, 8, 2019.

[71] Ania M Piotrowska, Jamie Hayes, Tariq Elahi, Sebastian Meiser, and George Danezis. The Loopix anonymity system. In *26th USENIX Security Symposium USENIX Security 17)*, pages 1199–1216, 2017.

[72] Michael K Reiter and Aviel D Rubin. Crowds: Anonymity for web transactions. *ACM transactions on information and system security (TISSEC)*, 1(1):66–92, 1998.

[73] Eric Rescorla and Tim Dierks. The Transport Layer Security (TLS) protocol version 1.3. RFC 1654, RFC Editor, July 1995. URL https://www.rfc-editor.org/rfc/rfc1654.txt.

[74] Charlie Savage. Chelsea Manning to be released early as Obama commutes sentence. *The New York Times*, 17, 2017.

[75] Michael S Schmidt and LM Steven. Panama law firm's leaked files detail offshore accounts tied to world leaders. *The New York Times*, 3, 2016.

[76] Amazon Web Services. Amazon EC2 instance types. https://aws.amazon.com/ec2/instance-types/, 2022. Accessed March 2022.

[77] Scott Shane. WikiLeaks leaves names of diplomatic sources in cables. *The New York Times*, 29:2011, 2011.

[78] Payap Sirinam, Mohsen Imani, Marc Juarez, and Matthew Wright. Deep fingerprinting: Undermining website fingerprinting defenses with deep learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1928–1943, 2018.

[79] David Smith. Trump condemned for tweets pointing to name of Ukraine whistleblower. *The Guardian*, 2019. URL https://www.theguardian.com/us-news/2019/dec/27/trump-ukraine-whistleblower-president. Accessed March 2022.

[80] The BBC. Putin critic Navalny jailed in Russia despite protests. URL https://www.bbc.com/news/world-europe-55910974. Accessed March 2022.

[81] The Freenet Project. Freenet, 2020. URL https://freenetproject.org/.

[82] The Invisible Internet Project. I2P anonymous network, 2020. URL https://geti2p.net/en/.

[83] The Tor Project. Tor metrics, 2019. URL https://metrics.torproject.org/.

[84] The Wall Street Journal. Got a tip? https://www.wsj.com/tips, 2020. Accessed March 2022.

[85] Yiannis Tsiounis and Moti Yung. On the security of ElGamal based encryption. In *International Workshop on Public Key Cryptography*, pages 117–134. Springer, 1998.

[86] Nirvan Tyagi, Yossi Gilad, Derek Leung, Matei Zaharia, and Nickolai Zeldovich. Stadium: A distributed metadata-private messaging system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 423–440, 2017.

[87] US Holocaust Memorial Museum. Röhm purge. *Holocaust Encyclopedia*, 2020. URL https://encyclopedia.ushmm.org/content/en/article/roehm-purge. Accessed March 2022.

[88] US Occupational Safety and Health Administration. The whistleblower protection program. https://www.whistleblowers.gov/, 2020. Accessed March 2022.

[89] US Securities and Exchange Commission. Office of the whistleblower. https://www.sec.gov/whistleblower, 2020. Accessed March 2022.

[90] Jelle Van Den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 137–152. ACM, 2015.

[91] Luis von Ahn, Manuel Blum, Nicholas J. Hopper, and John Langford. CAPTCHA: using hard AI problems for security. In *Advances in Cryptology - EUROCRYPT 2003, International Conference on the Theory and Applications of Cryptographic Techniques, Warsaw, Poland, May 4-8, 2003, Proceedings*, pages 294–311, 2003. doi: 10.1007/3-540-39200-9_18. URL https://doi.org/10.1007/3-540-39200-9_18.

[92] Von Spiegel Staff. Inside the NSA's war on internet security. *Der Spiegel*, 2014. URL https://www.spiegel.de/international/germany/inside-the-nsa-s-war-on-internet-security-a-1010361.html. Accessed March 2022.

[93] Lei Wang, Kazuo Ohta, and Noboru Kunihiro. New key-recovery attacks on HMAC/NMAC-MD4 and NMAC-MD5. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 237–253. Springer, 2008.

[94] Mark N Wegman and J Lawrence Carter. New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences*, 22(3):265–279, 1981.

[95] David Isaac Wolinsky, Henry Corrigan-Gibbs, Bryan Ford, and Aaron Johnson. Dissent in numbers: Making strong anonymity scale. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 179–182, 2012.

[96] Kim Zetter. Jolt in WikiLeaks case: Feds found Manning-Assange chat logs on laptop. *Wired*, 19 December 2011. URL https://www.wired.com/2011/12/manning-assange-laptop/. Accessed March 2022.

A The audit attack

While many broadcast systems claim privacy with a malicious server, they trade robustness to do so. When a message is *expected*, a server can act as if a user was malicious to prevent aggregation of their request, learning whether that user was responsible for the expected message. If a system aborts in such circumstances, it no longer has the claimed disruption-resistance property. Some systems such as Atom [55] and Blinder [2] solve this by using verifiable secret-sharing in an honest-majority setting; however, this can be costly in practice; others do not prevent this attack.

Express. Express is designed for private readers, but it can be trivially adapted for broadcast (see Sections 7 and 8). However, a malicious server can then exploit the verification procedure [41, Section 4.1] to exclude a user, changing their request to an invalid distributed point function. This excludes the message from the final aggregation, deanonymizing a broadcaster with probability at least $\frac{1}{(1-\epsilon)N}$ per round (where ϵ is the fraction of corrupted clients). Even with a few rounds, this can lead to a successful deanonymization of a broadcaster *without detection* (honest servers cannot tell if a server is cheating and therefore cannot abort the protocol).

Riposte. The threat model of Riposte does *not* consider attacks in which servers deny a write request. As a result, a malicious server can eliminate clients undetectably by simply computing a bad input to the audit protocol which causes the request to be discarded by both servers. While this attack can be mitigated by using multiple servers and assuming an honest majority (as in Blinder [2]), this weakens the threat model and reduces performance.

Application of BlameGame. The BlameGame protocol applies immediately to both Riposte and Express to address this audit attack by allowing (honest) servers to assign blame to either a client or a server if an audit fails. The only cost (as in Spectrum) is a slight increase in communication overhead which, importantly, is independent of the encoded message in the request (see Section 5.1).

B Large message optimization (multi-server)

In Section 5.1, we give a transformation from a 2-server DPF over a field $\mathbb{F}$ to a 2-server DPF over ℓ-bit bitstrings that preserves the auditability of the first DPF without increasing the bandwidth overhead proportionally. Here, we show a more

general transformation from n-server DPFs over a field $\mathbb{F}$ to n-server DPFs over a group $\mathbb{G}_y$ of a polynomially larger order. Our transformation uses a seed-homomorphic pseudorandom generator (PRG) [12].

Definition 2 (Seed-homomorphic Pseudorandom Generator). *Fix groups $\mathbb{G}_s, \mathbb{G}_y$ with respective operations $\circ_s$ and $\circ_y$. A seed-homomorphic pseudorandom generator is a polynomial-time algorithm $G : \mathbb{G}_s \to \mathbb{G}_y$ with the following properties:*

- Pseudorandom. *G is a PRG: $|\mathbb{G}_s| < |\mathbb{G}_y|$, with output computationally indistinguishable from random.*
- Seed-homomorphic. *For all $s_1, s_2 \in \mathbb{G}_x$, we have $G(s_1 \circ_s s_2) = G(s_1) \circ_y G(s_2)$.*

Let $\mathbb{G}$ be a group over a field $\mathbb{F}$ and in which the decisional Diffie-Hellman (DDH) problem [10, 11, 40] is assumed to be hard. Fix some DPF with messages in $\mathbb{F}$. We saw in Section 4.2 how to implement anonymous access control for such DPFs. Let $G : \mathbb{F} \to \mathbb{G}_y$ be a seed homomorphic PRG where $\mathbb{G}_y$ is over $\mathbb{F}$. Boneh et al. [12] give a construction of such a PRG for $\mathbb{G}_y = (\mathbb{G})^L$ from the DDH assumption in $\mathbb{G}$.

Then, the larger DPF key for a message m is a DPF key k_1 for a random value $s \in \mathbb{F}$, a DPF key k_2 for $1 \in \mathbb{F}$, and a "correction message" $\bar{m} = m \circ_y G(s)^{-1}$ (each key has the *same* correction message). For a zero message, the larger DPF key is two DPF keys k_1, k_2 for $0 \in \mathbb{F}$ and a random correction message $\bar{m}$.

To evaluate the DPF key, the server computes $s \leftarrow$ DPF.Eval(k_1), $b \leftarrow$ DPF.Eval(k_2), and $(\bar{m})^b \circ_y G(s)$. If $s = 0$, then combining the DPF keys gives $(\bar{m})^0 \circ_y G(0) = 1_{\mathbb{G}_y}$. Otherwise, we get $(\bar{m})^1 \circ_y G(s) = m$.

To perform access control for the larger DPF, perform access control for k_1 and k_2 and then also check for the equality of the hashes of $\bar{m}$. We note this construction does not yield a new DPF, but does add authorization to a large class of existing DPFs.

C BlameGame

C.1 Verifiable Encryption

BlameGame (Section 4.3) uses a verifiable encryption scheme [20], which allows a prover to decrypt a ciphertext c and create a proof that c is an encryption of a message m. We formalize these schemes below:

Definition 3 (Verifiable Encryption). *A verifiable public-key encryption scheme $\mathcal{E}$ consists of (possibly randomized) algorithms* Gen, Enc, Dec, DecProof, VerProof *where* Gen, Enc, Dec *satisfy IND-CPA security and* DecProof, VerProof *satisfy the following properties:*

- Completeness. *For all messages $m \in \mathcal{M}$,*

$$\Pr\left[\begin{array}{l} (\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{Gen}(1^\lambda); \\ c \leftarrow \mathsf{Enc}(\mathsf{pk},m); \\ (\pi,m) \leftarrow \mathsf{DecProof}(\mathsf{sk},c); \\ \mathsf{VerProof}(\mathsf{pk},\pi,c,m) = \mathsf{yes} \end{array}\right] = 1,$$

where the probability is over the randomness of Enc.

- Soundness. *For all PPT adversaries $\mathcal{A}$ and for all messages $m \in \mathcal{M}$,*

$$\Pr\left[\begin{array}{l} (\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{Gen}(1^\lambda); \\ c \leftarrow \mathsf{Enc}(\mathsf{pk},m); \\ (\pi,m') \leftarrow \mathcal{A}(1^\lambda,\mathsf{pk},\mathsf{sk},c); \\ \mathsf{VerProof}(\mathsf{pk},\pi,c,m') = \mathsf{yes} \end{array}\right] \leq \mathsf{negl}(\lambda)$$

for negligible function negl(λ)*, where the probability is over the randomness of* Enc *and $\mathcal{A}$.*

We note that many public key encryption schemes (e.g., El-Gamal [40]) satisfy Definition 3 out-of-the-box and can be used to instantiate BlameGame.

C.2 BlameGame security

The BlameGame protocol must be *sound* and *private*.

Soundness. BlameGame is *sound* if no honest client or server will ever be blamed:

1. For all honest commitments C_i, no probabilistic polynomial-time (PPT) adversary can create a request share τ_i and proof of decryption π_i such that the BlameGame "Assigning blame" step (Section 4.3) blames the client when run with (π_i, τ_i, C_i).

2. No PPT adversary can create commitments C_i such that an honestly-created request share τ_i and proof of decryption π_i will result in blaming the server after running the BlameGame "Assigning blame" step.

Privacy. The privacy requirement of BlameGame is similar to that of Spectrum. Specifically, the commitments C_i must not reveal any information about the request to any subset of servers. Formally: for randomly sampled pairs of keys pk_i and sk_i (for $i \in \{1,\ldots,n\}$), and all proper subsets $I \subset \{1,\ldots,n\}$, the following distributions are computationally indistinguishable:

$$\{(\mathsf{pk}_i,\mathsf{sk}_i)\ \forall i \in I,\ C_i\ \forall i \in \{1,\ldots,n\}\} \approx_c$$
$$\{(\mathsf{pk}_i,\mathsf{sk}_i)\ \forall i \in I,\ C_i'\ \forall i \in \{1,\ldots,n\}\}$$

where the C_i are created by honestly encrypting request shares corresponding to a cover request by a subscriber and the C_i' are created by honestly encrypting shares corresponding to any valid write generated by a broadcaster.

We note that BlameGame does **not** require any privacy properties during blame assignment, as it may reveal the request for the purpose of assigning blame.

We now show that BlameGame achieves these properties:

Proof. We prove each property in turn.

Soundness (honest client). Suppose, toward contradiction, that there exists a PPT adversary $\mathcal{A}$ that generates some request shares τ_i and proof of decryption π_i such that BlameGame blames the client. This means that (1) the decryption proof verification succeeds, and (2) running the audit with the request shares failed. By property (1), we can assume that τ_i is a correct decryption of C_i and π_i is a valid proof of decryption; otherwise, $\mathcal{A}$ breaks the soundness property of the verifiable encryption scheme. However, we know that running the audit with the given request shares will pass, because (by assumption) they were created honestly by the client. This is a contradiction.

Soundness (honest server). Let τ_i be a set of request tokens such that the Spectrum audit fails when run with τ_i. Suppose, toward contradiction, that some client creates commitments $C_1, \ldots, C_n$ for $\tau_1, \ldots \tau_n$ such that the BlameGame "Assigning blame" step blames some server (instead of the client, as required). Then, it holds that either (1) the proof of decryption failed, or (2) the audit performed by the servers over the decrypted requests passes. However, if (1) is true (the proof of decryption failed), then the completeness property of the verifiable encryption scheme does not hold (because the request share and proof of decryption are generated honestly by the server). Therefore, we are left with (2); the audit performed by the servers over the decrypted request shares passes. However, this isn't true (by assumption) if the client is malicious. Hence, we have a contradiction.

Privacy. For all honest broadcasters, privacy is guaranteed with probability $\frac{L}{N \cdot (1-\epsilon)}$ where ϵ is the fraction of corrupted clients. If the first audit fails but the second audit (generated from the decrypted requests) passes, then privacy follows from the analysis of Spectrum and privacy of the audit therein. If the second audit fails, then the request is revealed to both servers for inspection (in order to adequately assign blame). However, predicated on the revealed request being generated correctly (since we are interested in when an *honest* broadcaster gets deanonymized), the protocol aborts if the second audit fails (an honest broadcaster would have encrypted the request correctly). In this case, all servers see the request which deanonymizes the client. Thus, for a fraction of corrupted clients ϵ, the probability that the malicious server chooses the correct request to tamper with before being aborted is $\frac{L}{N \cdot (1-\epsilon)}$. □

Spectrum (with BlameGame) achieves our desired security properties: a malicious client cannot cause disruption, and a malicious server cannot deanonymize a broadcaster. Because BlameGame is sound, if all servers are honest then Spectrum does not abort (because either the audit passes, or BlameGame blames the client); this prevents disruption due to audit failure. The second property follows from the privacy of BlameGame.

Donar: Anonymous VoIP over Tor

Yérom-David Bromberg, Quentin Dufour, Davide Frey
Univ. Rennes - Inria - CNRS - IRISA, France

Etienne Rivière
UCLouvain, Belgium

Abstract

We present DONAR, a system enabling anonymous VoIP with good quality-of-experience (QoE) over Tor. No individual Tor link can match VoIP networking requirements. DONAR bridges this gap by spreading VoIP traffic over *several* links. It combines active performance monitoring, dynamic link selection, adaptive traffic scheduling, and redundancy at no extra bandwidth cost. DONAR enables high QoE: latency remains under 360 ms for 99% of VoIP packets during most (86%) 5-minute and 90-minute calls.

1 Introduction

Tor [20] is by far the largest anonymization network with over 6,000 relay nodes distributed worldwide. Tor has been very successful for applications such as web browsing with, e.g., TorBrowser, but is generally considered inadequate for latency-sensitive applications [31,66]. Voice-over-IP (VoIP) is one such application that has become the *de facto* solution for global voice calls. Being able to deploy VoIP over Tor would immediately benefit privacy-conscious users by enabling simple, efficient, and safe voice communication answering two objectives: (i) protecting the content of the communication from adversaries, i.e., using end-to-end encryption, and (ii) hiding metadata and in particular the identity of communicating partners. Metadata may, indeed, be used to infer private information, e.g., uncovering a journalist's sources [27] or illegally gathering information about employees [60].

Providing good-quality interaction between VoIP users, i.e., a good Quality-of-Experience (QoE), requires good network Quality-of-Service (QoS) and in particular low and stable latency [28, 35, 68], as we detail in **Section 2**. This comes in tension with the way Tor is designed [31,66]: Tor links[1] implement multi-hop communication for TCP traffic using *onion routing* over pre-established circuits formed of several relays, which leads to high and unstable latencies. Surprisingly, Sharma *et al.* [70] recently posited that using Tor *as is* would

---

[1]We use in this paper the generic term *link* to denote the unidirectional TCP channel that is exposed to applications by the Tor client.

be sufficient to obtain the stable and low latencies required by high-QoE VoIP. This statement is, unfortunately, incorrectly grounded. Three biases in their analysis led to this conclusion: (1) they only consider average latencies, while VoIP QoE is primarily determined by tail latency (99th percentile with a standard codec) [57], (2) they measure performance for only 30 seconds, a much shorter duration than an average call [33], and (3) they only consider the case of one-way anonymity, i.e., when the callee is not anonymous. We present in **Section 3** our analysis of Tor links' performance considering these elements and conclude that the use of a Tor link *as is* does not, in fact, allow VoIP with sufficient QoE.

TorFone [24] attempts to overcome Tor latency issues by duplicating VoIP traffic over two statically chosen links. However, even if going in the right direction, TorFone's strategy turns out to be ineffective due to the large variability of link performance over time, as we demonstrate in **Section 6**.

Alternative anonymization networks targeting voice communication were also proposed recently, e.g., Herd [50] and Yodel [49]. These systems, however, are yet to be deployed and need to reach a sufficient scale to be efficient, i.e., to provide sufficient bandwidth for a large number of geographically-distributed users.

Motivations. We are interested in providing VoIP support over a *readily-available* anonymization network. More specifically, we target a deployment using (1) legacy VoIP applications and (2) the existing, unmodified Tor network. We do not wish to propose design changes to Tor, or a novel anonymity network [49, 50, 64, 73, 76], and neither do we want to overcome Tor's existing security flaws. We believe that these lines of work are, in fact, orthogonal to our own.

While our observation of the performance of Tor (presented in **Section 3**) confirms that a *single* Tor link cannot provide the stable and low latency required by high-QoE VoIP, it also allows us to make a case for dispatching traffic over *multiple* links. Unlike TorFone, our strategy multiplexes traffic over a *dynamically selected* set of Tor links using a smart scheduling mechanism. Our motivation is that the use of multiple dynamically and adequately chosen links, together with controlled

and smart content redundancy, can mask the transient faults and latency spikes experienced by individual links.

Contributions. We present the design and implementation of DONAR, a user-side proxy interfacing a legacy VoIP application to the existing Tor network (**Section 4**).

DONAR enforces *diversity* in the paths used for transmitting VoIP packets, i.e., using distinct Tor links. In addition, it leverages *redundancy* by sending the same VoIP packet several times using different links. This redundancy does not, in fact, add additional bandwidth costs for the Tor network beyond those incurred by the setup and maintenance of these multiple links. We leverage, indeed, the fact that Tor only transmits 514-Byte cells over the network to protect users against traffic analysis [53, 59]. DONAR takes advantage of the available padding space to re-transmit previous VoIP packets. Diversity and redundancy mask the impact of the head-of-line blocking implied by the TCP semantics of Tor links, whereby an entire stream of packets may get delayed by a single belated one.

DONAR builds on the following key technical components:

- The *piggybacking* of VoIP packets in the padding space of Tor cells enables redundancy without incurring additional bandwidth costs on the Tor network.

- A *link monitoring* mechanism observes and selects appropriate links, switching rapidly between them when detecting performance degradation.

- Two *scheduling* strategies for selecting links when transmitting VoIP packets enable different tradeoffs between cost and robustness.

We further analyze in **Section 5** how attacks on Tor can affect the security properties of DONAR. In particular, we discuss how different DONAR configurations implement different tradeoffs between Quality-of-Experience and security.

We evaluate DONAR over the Tor network and present our findings in **Section 6**. We use VoIP-traffic emulation as well as the off-the-shelf `gstreamer` [26] VoIP client using the OPUS [14] audio codec. We assess the performance of DONAR against the VoIP requirements detailed in Section 2 and compare it with the approach followed by TorFone [24]. Our results show that DONAR, using alternatively 6 out of 12 carefully monitored and dynamically selected onion links, achieves high QoE with latency under 360 ms and less than 1% of VoIP frame loss for the entire duration of a large number (86%+) of 5-minute and 90-minute calls, with no bandwidth overhead for its optimized configuration (i.e., alternate sending over different links) and an overhead similar to that of TorFone for its default configuration.

We detail related work and conclude in **Sections 7 and 8**.

2 VoIP networking requirements

DONAR aims at Providing good Quality-of-Experience (QoE) for anonymous VoIP while limiting the costs imposed on the

Metric	Objective
Dropped calls rate	$\leq 2\%$ for 90-minute calls
Packet loss rate	$\leq 1\%$
Bandwidth	≥ 32 kbps (4.3 kB/s)
One way delay (99th perc. *ideal*)	≤ 150 ms - T_{frame} - T_{buffer}
One way delay (99th perc. *max*)	≤ 400 ms - T_{frame} - T_{buffer}

Table 1: VoIP network performance requirements, following the recommendations of the International Telecommunication Union [35] and applying them to the OPUS codec [74, 75].

Tor infrastructure. We base our analysis of QoE requirements on recommendations by the International Telecommunication Union (ITU) [35–37]. The ITU defines good QoE as the combination of the following guarantees: (1) uninterrupted calls, (2) good voice quality, and (3) support for interactive conversations. We analyze these requirements and derive our network QoS objectives, summarized in Table 1.

VoIP protocols. VoIP requires two types of protocols. A signaling protocol such as the Session Initiation Protocol (SIP) [67] makes it possible to locate a correspondent and negotiate parameters for the communication. The signaling protocol only impacts QoE with delays upon the establishment of the call. When the call is established, a protocol such as the UDP-based Real-time Transport Protocol (RTP) [69] is used to transmit VoIP audio frames encoded using a codec, whose configuration is negotiated by the signaling protocol. QoE is primarily impacted by this codec and its ability to deal with hazards in network QoS, as we detail next.

Impact and choice of the audio codec. Bandwidth, latency, or maximum packet loss requirements depend on the audio codec used by the VoIP application. We base our analysis on the state-of-the-art open audio codec OPUS, which we also use in our evaluations. OPUS is a widely-used, loss-tolerant audio codec developed by the Xiph.Org Foundation and standardized by the IETF [74, 75]. It targets interactive, low-delay communication and computational efficiency. OPUS has been consistently ranked in comparative studies as the highest-quality audio format for low and medium bit rates [32, 41]. We emphasize that our analysis would be similar for other open codecs, e.g., the Internet Low Bit Rate Codec (iLBC) [4] or Xiph.Org Foundation's former codecs Vorbis [7] and Speex [30].

First guarantee: *no call interruption*. A call interruption is the most impacting event on user-perceived QoE. The ITU does not provide a recommendation for general networks but recommends at most 2% dropped calls for VoIP over 4G [37]. We adopt the same goal but need to define a time span on which to evaluate this metric. Holub *et al.* [33] provided us with a dataset of more than 4M call durations (Figure 1). Its analysis confirms that call duration follows a log-normal distribution considered as standard for voice calls. We observe an average call duration slightly above 3 minutes, with 90% of
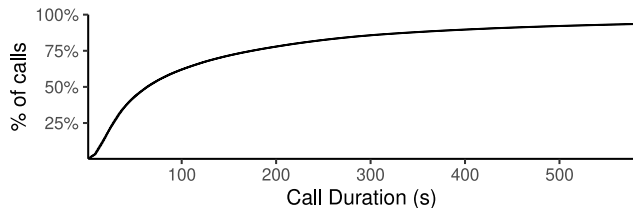
Figure 1: Call Duration ECDF on a 4M calls dataset provided by Holub *et al.* [33] zoomed on the first 10 minutes.

Figure 2: Structure of a Tor link with onion services.

calls lasting less than 10 minutes. However, we still observe 1,040 calls lasting for 90 minutes or more which is characteristic of the long tail of the distribution. As this value matches the limitation set by major representative carriers [34,77], we evaluate reliability for calls over this duration.

Second guarantee: *good voice quality*. Users want to clearly hear their communication partners. Voice quality depends both on the bitrate being used and the amount of packet loss: (1) Listening tests with OPUS [14,32] concluded that a bitrate of 32 kbps is sufficient to offer a sound quality that test users cannot distinguish from a reference unencoded version of the recording. We set, therefore, this bitrate as the minimum required bandwidth that we must offer to the VoIP application. (2) OPUS provides two mechanisms to mask the impact of lost packets: a domain-specific one, named Packet Loss Concealment (PLC) and a generic one, via redundancy, named Forward Erasure Coding (FEC)[2] [72]. Han *et al.* [28] studied the perceived quality of a call on various packet rates. This study shows that while PLC compensates for packet loss, the perceived voice quality nonetheless decreases quickly: a 1% packet loss is essentially unnoticed, while 10% packet loss results in usable but degraded call conditions. Based on these results, we set as a requirement a packet loss of at most 1%.

Third guarantee: *interactive conversations*. In addition to an uninterrupted and good-quality voice signal, users of voice calls expect to be able to exchange information interactively, e.g., be able to seamlessly synchronize on when to stop and start talking in a conversation.

Interactivity primarily depends on latency [68]. The ITU published recommendation G.114 [35] on mouth-to-ear latency in voice calls. This recommendation indicates that a delay below 150 ms is unnoticeable for users, compared to a direct voice conversation. We set, therefore, this value as our ideal latency. On the other hand, the recommendation stipulates that delays must remain below 400 ms to make an interactive call possible under good conditions. Higher latencies result in synchronization difficulties and significantly reduce user-perceived QoE. We set this threshold of 400 ms as our maximum acceptable mouth-to-ear latency.

We emphasize that the actual network latency for transmitting VoIP frames is only a subset of mouth-to-ear latency.
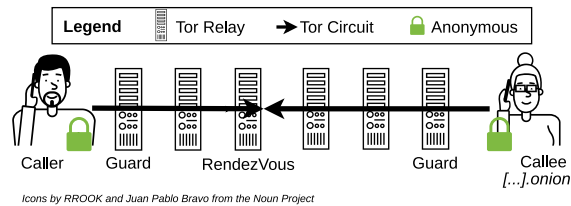
Additional latency is introduced by (1) audio capture and playing, (2) packetization, and (3) buffering. Once digitized, audio is encapsulated, every T_{frame} ms, into frames that will form a packet. OPUS enables configurable values for T_{frame} from 2.5 to 60 ms.

We consider an ideal jitter-buffer model similar to the one by Moon *et al.* [57]. This model delays all frames by the maximum or nth percentile of the observed latency and allows frame drops. Moon *et al.* [57] and others [44, 52] have proposed jitter-buffer implementations performing close to this theoretical optimum. Therefore, we consider T_{buffer}, the unnecessary delay added by a wrong jitter-buffer configuration as negligible. Finally, as we allow a 1% frame drop, we consider the 99th latency for our mouth-to-ear delay constraints.

3 VoIP over Tor: How bad is it?

Tor [20] is a large-scale network that enables users to access remote resources anonymously. Tor relies on *onion routing*: it relays traffic through *circuits* consisting of at least two relays (three by default) chosen from more than 6,000 dedicated nodes. The first relay in a circuit is known as the *Guard*. The Tor client chooses a small set of n (by default[3], $n = 2$) possible guards. Thereafter, it builds circuits by using one guard from this set, choosing the remaining relays randomly from the list of all available relay nodes.

Tor enables both connections to the regular Internet (referred to as *Exit*) and to other Tor users (referred to as *Onion Services*). In contrast to the *Exit* mode, *Onion Services* provide two-way anonymity by default. The Tor client on the caller's side connects to an anonymous onion service (set up by the callee's Tor client). In doing so, it creates a Tor route, i.e., the concatenation of two Tor circuits, one from the caller to a rendezvous relay, and another from the callee to the same rendezvous relay. In this paper, we use the term *link* to refer to the TCP connection over this route that the Tor client exposes to the application. Figure 2 illustrates a Tor link based on an onion service used for transmitting VoIP frames.

Tor seeks to prevent adversaries from inferring communicating parties. To this end, at least one relay in the route should lie in an administrative domain that the adversary cannot observe. Furthermore, to prevent traffic analysis attacks, Tor only sends fixed-sized messages between relays, in the

---

[2]We configure OPUS to use only the former, as DONAR already enables redundancy mechanisms that are specific to the Tor network.

[3]While Tor advertises using $n = 1$ by default, it effectively uses $n = 2$ [62].
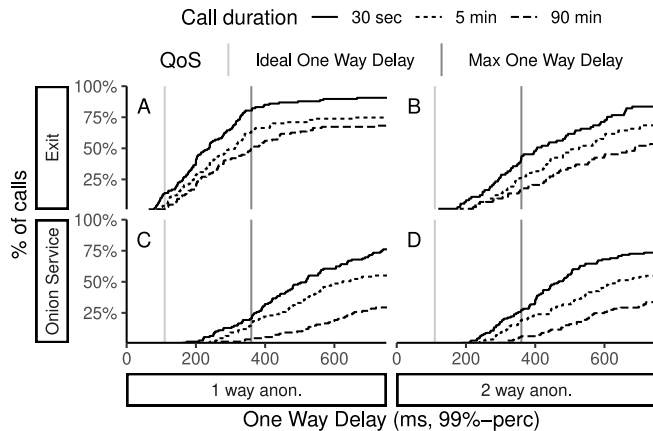
Figure 3: VoIP over a single Tor link: Evolution of the one-way delay's 99th percentile according to connection-link type and call duration.

form of 514-Byte *cells* [53, 59]. When a packet being transmitted over a Tor link is less than 514 Bytes in size, the Tor client pads it with random data.

3.1 Evaluation of Tor links' QoS

Tor is often described as a *low-latency* anonymization network. Its TCP streams over pre-established circuits enable, indeed, lower latency than anonymization networks where the relays for each message in a stream are chosen independently [12, 73, 76]. The latency of links in Tor, and in particular their stability, is however known to be unpredictable, which made several authors doubt Tor's ability to support low-latency applications such as VoIP [31, 66].

In this section, we report on our own experimental evaluation of the network QoS of Tor links. We confirm the observation made by other authors that a single Tor link is unsuitable for VoIP networking requirements as defined in the previous section. However, these measurements allow us to make the case for the foundational design choice in DONAR: using several, dynamically selected links.

We consider the following metrics: the connection stability, the variability of one-way latency, and the predictability of high latency from prior measurements. We use a load injector with varying packet-sending rates and, in order to measure one-way latency, a stub communication endpoint located on the same machine. The injector and the stub use two separate instances of the Tor client in its default configuration and create circuits independently. All reported experiments were conducted in January 2021.

Connection links. We start by analyzing how the two Tor modes, *Exit* and *Onion Services*, perform in terms of tail latency. Each of these modes can be declined in links providing either one-way or two-way anonymity. *Exit* links provide one-way anonymity by default but we can mimic two-way anonymity by making both caller and callee access the same

public VoIP server. *Onion-Service* links provide two-way anonymity by default but we can reduce the number of relays and keep only one-way anonymity. We use the HIDDENSER-VICESINGLEHOPMODE feature in the Tor daemon to achieve one-way anonymity over Onion Services.

Considering these 4 configurations, we simulated VoIP calls lasting 30 seconds, 5 minutes, and 90 minutes. The simulation strictly follows the requirements presented in **Section 2**. For each combination of configuration and call duration, we made 64 calls and present the results in Figure 3.

We start our analysis by focusing on Figure 3.A as it features the configuration on which Sharma *et al.* [70] base their claim that Tor links are suitable *as is* to support VoIP. With 37% of unacceptable calls (resp. 50%) for 5-minute (resp. 90-minute) calls, we argue the opposite. We identified three reasons explaining why our analysis differs. (1) They do not account for T_{frame} in their analysis. Since we use $T_{frame} = 40$ ms, our max acceptable latency is 360 ms. (2) They consider average latencies instead of the 99th percentile of their distribution. While we obtain similar average latencies, considering tail latency shows that 20% of calls suffer from unacceptable delays, even for short 30-second calls. (3) They consider only such 30-second calls when the average call duration is 3 minutes and when a significant share of calls last up to 90 minutes. Measuring links over a longer timespan shows, in fact, that latencies tend to increase with call duration.

Comparing the different configurations we observe that, in fact, no link type offers acceptable delays. We note (Figure 3.{B,D}) that the latency benefits from using the *Exit* mode mostly vanish when considering 2-way anonymity. Using one-way anonymity with the *Onion Service* mode (Figure 3.C) does not seem to improve tail latency; we presume this is because this feature is still experimental.

Moreover, not all link types are equal: using *Exit* links has two drawbacks. First, it requires the last relay of the circuit to hold the *Exit* tag. As *Exit* links can send data on the regular Internet, the last relay is particularly sensitive: only 25% of the relays accept to have this position. From the user's perspective, this situation eases de-anonymization attacks and, by limiting the scalability of the network, also harms performances. Moreover, using *Exit* links requires relaying traffic through an ad-hoc public server that must be trusted (e.g., Sharma *et al.* [70] use Mumble and Freeswitch PBX).

Considering that (i) no link type over Tor enables VoIP, and (ii) the *Exit* mode has severe limitations, we choose to focus solely on leveraging *Onion-Service* links to provide anonymous voice calls in the rest of this paper.

Connection stability. We evaluate the reliability of each Tor link type over our longest considered call duration (90 minutes). Figure 4 reports the cumulative rate of failed links (i.e., for which packets are no longer transmitted) as a function of time. After 10 minutes, all link types exhibit failure rates of at least 4%. The rate rises to between 7% and 16% after one hour. The failure difference between link types seems
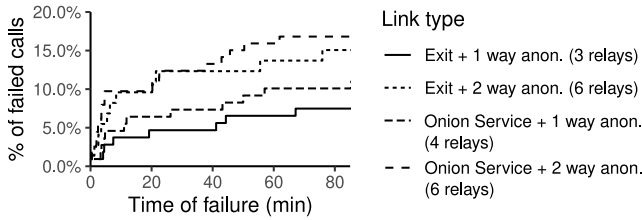
Figure 4: Failed Tor links over time.
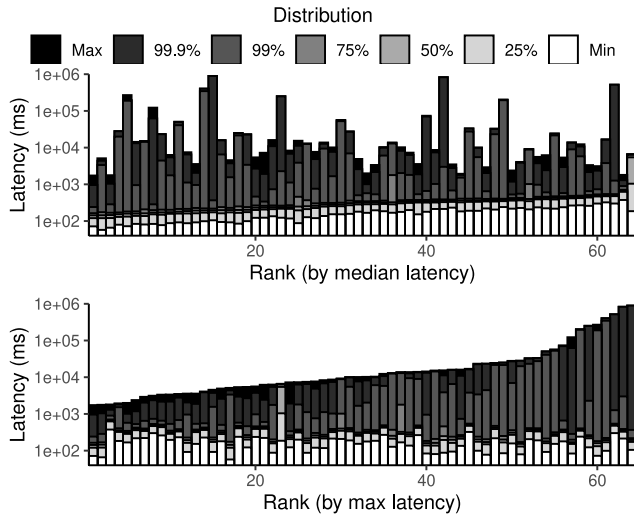
Figure 5: Tor links' latency distribution at 25 packet/sec ordered by median (top) and max (bottom) latency.

to be correlated with their number of relays: the more there are relays, the more likely failures are. None of the available links satisfies our QoS requirements: we need a solution that overcomes link breakage and allows calls to continue seamlessly.

Predictability of high latencies. The previous experiment shows that the distribution of latency across multiple links is highly skewed. We now evaluate if this skew results from a large number of poorly performing links with a few, identifiable, good links, or if any link can experience periodic latency bursts. Figure 5 presents the one-way-latency distribution for each of the 64 links, ranked by median latency (top) or max latency (bottom). There is no clear relationship between the general performance of a link and the occurrence of latency spikes. The maximal latency does not seem to depend much on the rest of the distribution and can reach very high values in all cases (often 3 times higher than the 75th percentile)[4]. We refer to these high latency periods as *latency spikes* in the rest of this paper.

Discussion. Our experiments confirm the general unpredictability of the performance of Tor links. Due to Tor's ex-

---

[4]This unpredictable performance is confirmed, in fact, by a blog post by the Tor project [63]. We quote: "*While adding more relays to the network will increase average-case Tor performance, it will not solve Tor's core performance problem, which is actually performance variance.*".

clusive support for TCP[5], latency spikes for a single packet result in high latency for all following packets, delayed to be delivered in order—a phenomenon referred to as *head-of-line blocking*.

We observe, however, that the number of relays correlates with the probability of networking problems: larger numbers of relays are associated with higher failure rates or with latency spikes. We also note that most links provide good performance for a fraction of their use time, and failures across links do not seem to be correlated. As a result, we make the case for using multiple links, benefiting from periods of good performance, and quickly switching links when experiencing latency spikes.
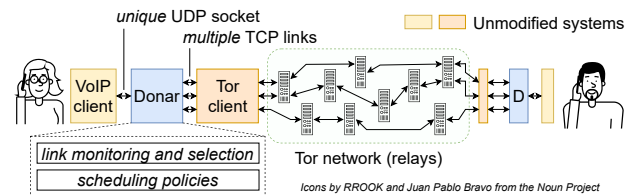
Figure 6: DONAR plays the role of a proxy between an unmodified VoIP application and the unmodified Tor client.

4 Donar: Enabling VoIP over Tor

DONAR operates as a proxy between a VoIP application and the Tor client, as illustrated in Figure 6. It does not require modifying either of the two systems. DONAR runs without any specific privileges; it only offers a UDP socket to the VoIP application's RTP protocol and opens TCP sockets (links) with the local Tor client. In conformance with our objective to make anonymous VoIP available with *readily-available* systems, we do not require the deployment of external support services. In particular, DONAR does not rely on the SIP signaling protocol but leverages instead Tor onion addresses to establish communication without leaking metadata about communicating parties.

Redundancy by piggybacking. DONAR leverages the fact that Tor only transmits data in the form of fixed-sized cells. Setting OPUS to the target bitrate of 32 kbps and using a sending period of 40 ms results in 172-Byte frames. The Tor client pads the remaining space with random data to reach a cell size of 514 Bytes. DONAR leverages, instead, this space to re-send the previous frame without changing the necessary bandwidth requirements[6]. Naturally, a redundant frame must

---

[5]TCP maps well to an efficient implementation of onion routing, i.e., it makes it possible to know when to create and dispose of circuits and it avoids the presence of packets that are untied to an existing circuit. UDP would also pose security challenges, e.g., enable DDoS attacks. The designers of Tor have clearly dismissed any support of UDP in Tor in the future [56].

[6]We are not limited to this configuration, and only require that the size of the frames emitted by the codec be less than half the available space minus the Tor headers (8 Bytes) and DONAR metadata (38 Bytes in the default configuration), i.e., less than 233 Bytes.

be sent on a different link than the first copy, to avoid head-of-line blocking between replicas. While redundant frames are subject to an additional T_{frame} latency (40 ms in the presented configuration), our rationale is that this latency combined with that of the link itself will still be lower than that of a link experiencing a latency spike. We detail next how we effectively enable link diversity.

Link Diversity. DONAR leverages multiple Tor Onion-Service links to multiplex traffic in two complementary ways. First, it spreads frame copies onto different links. This prevents packets containing subsequent frames from being subject to the same latency spike thereby arriving too late in a burst at the destination. This also lowers the load on each individual link (resulting, as shown in Section 3, in better availability). Second, DONAR ensures that the first and the second (redundant) copy of a given frame always travel on different links.

Enabling diversity requires (1) maintaining a set of open links and monitoring their performance; and (2) implementing a scheduling policy for selecting appropriate links for new packets. In the following, we detail these two aspects (§4.1 and §4.2) and complete the description of DONAR by detailing how calls are established (§4.3).

4.1 Link monitoring and selection

DONAR opens and monitors a set of Tor links and associates them with scores reflecting their *relative* latency performance. We start by detailing how latency scores are collected at the local client's side, and why they must also be collected from the remote client. We motivate our choice to classify links in performance groups, and how we dynamically select links in these groups throughout a call.

Measuring latency. Measuring transmission delays for packets sent over Tor is not straightforward. The RTP protocol uses UDP and does not send acknowledgments. We do not wish to add additional acknowledgment packets over Tor to measure round-trip times, as their padding in 514-Byte cells would double bandwidth consumption.

Rather than attempting to measure the *absolute* latencies of links, we leverage the use of multiple links to approximate their *relative* latency performance. Measures of performance are continuously collected on both sides of the communication, which we denote as node A and node B in the following. Local *aggregate* measures are then computed over a time window of duration w. We explore the impact of durations ranging from 0.2 to 32 seconds in our evaluation.

We base our measurements on an *out-of-order* metric for VoIP frames. This metric denotes, for an incoming frame f with sequence number i, the number of frames received *before* f with a higher sequence number than i. As TCP delivers packets in order, these frames necessarily travel on different links. For instance, if node A receives frame f with sequence number i from node B on link l after receiving frames with sequence numbers $i+1$, $i+2$, and $i+3$ on other links, we associate an out-of-order metric of 3 to frame f.

The local calculation of the out-of-order metric also applies to *missing* frames. Node A is aware of any *missing* frame f_m with a sequence number $i_m < i_c$ where i_c is the largest sequence number among all the frames received from node B. However, since the decision on which link a packet is sent is made by node B, it is not possible for node A to assign f_m's measurement to a specific link. To solve this problem, we include, in the DONAR headers in each packet, the list of links used for sending the latest n frames, where n is the maximum number of used links.

Nodes A and B must share their local aggregate measures to enable fast detection of latency spikes. Node A's local information about a link l approximates, indeed, the one-way latency from B to A, but not from A to B. Our experimental evaluation has shown that one-way latencies are highly consistent in both directions of a link, making node A's local estimation a good approximation also for the latency from A to B. However, this local approximation may be missing if the link has not been used recently by B to send packets to A. We alleviate this problem by embedding, in the DONAR metadata sent with each packet, the local aggregate measures for links that have been measured recently. Node A computes a final array of measures that include, for each link, either (1) the local aggregate measure only, if no remote aggregate was received; (2) the remote aggregate only, if the link was not recently used by B to send data to A; or (3) the average of these two measures if the link was used in both directions.

Link selection. Every w seconds, DONAR sorts links in decreasing order of aggregated scores over the last period and assigns links to three groups. The $L_{1\text{ST}}$ (first-class) group contains the $n_{1\text{ST}}$ *fastest* links. The $L_{2\text{ND}}$ (second-class) group contains the $n_{2\text{ND}}$ following links. Typically, we use the same number of links in the two groups, i.e., $n_{1\text{ST}} = n_{2\text{ND}}$. Finally, the remaining $n_{\text{INACTIVE}} = n_{\text{LINKS}} - n_{1\text{ST}} - n_{2\text{ND}}$ slowest links are assigned to the L_{INACTIVE} group.

The rationale for this classification is as follows. Links in the L_{INACTIVE} group experience sub-par performance and must remain idle. Links in the $L_{1\text{ST}}$ group have good performance and are invaluable in allowing fast delivery of VoIP packets. However, the number of good-performing links is limited at a given point in time, and using them systematically bears the risk of overloading them, resulting in lower performance and reliability (§3). Links in the $L_{2\text{ND}}$ group are less performant, but remain usable, and can reduce this risk of overload.

Links opening and maintenance. DONAR uses standard operations of the Tor client to open links. It lets the client select relays according to Tor rules. The client allows users to parameterize the number of used guard relays, as well as the length of the links (number of relays). DONAR leverages these parameters to enable different security/performance tradeoffs. We defer the discussion of strategies for setting these values and their security implications, to Section 5.

When starting a call, DONAR opens $n_{\text{LINKS}} = n_{1\text{ST}} + n_{2\text{ND}} + n_{\text{INACTIVE}}$ links and assigns them randomly to the three groups. When the Tor client notifies a link failure, DONAR simply requests a new link and assigns it to the L_{INACTIVE} group.

Links in the L_{INACTIVE} group will not be monitored locally. Some of these links may be associated with a remote score, but others will not be monitored on either side of the call. To enable *all* links to be monitored periodically, we implement a promotion and demotion mechanism between the $L_{2\text{ND}}$ and L_{INACTIVE} groups. When assigning links to groups at the end of a period of *w* seconds, DONAR picks the worst-performing link from the $L_{2\text{ND}}$ group and demotes it to the L_{INACTIVE} group. In return, it promotes to $L_{2\text{ND}}$ the link from the L_{INACTIVE} group that has been unused for the longest time.

4.2 Scheduling policies

The DONAR scheduler receives UDP RTP packets containing a single frame from the VoIP application. It first implements redundancy by using the pad space to piggyback packets that were previously sent on different links, then adds the necessary metadata, and finally creates a TCP packet to be sent onto one or two links from the $L_{1\text{ST}}$ and/or $L_{2\text{ND}}$ groups.

DONAR's default scheduling policy is named ALTERNATE. It sends each new packet to a *single* link. In doing so, it *alternates* between links from the $L_{1\text{ST}}$ and $L_{2\text{ND}}$ groups. This complies with the requirement to send the first and redundant copies of a frame on different links. DONAR picks the links from each group using a round-robin policy, thereby complying with the requirement of maximizing diversity.

We implement a second policy named DOUBLE-SEND. As the name implies, this policy selects *two* links—one from $L_{1\text{ST}}$ and one from $L_{2\text{ND}}$—for sending each new packet. Each frame is received four times: two as a primary copy, and two as a duplicate. This policy doubles the required bandwidth but has a higher chance to select a fast link for the primary copy of a frame, thereby reducing the risk of delivering the frame with an additional delay of T_{frame}. We note that the resulting bandwidth is the same as for TorFone [24]'s Duplication mode, which systematically sends VoIP packets onto the same two links.

4.3 Establishing communication

DONAR leverages Tor's mechanisms to allow callers and callees to establish a connection anonymously. Following our design goal of using only readily-available systems, we do not require the deployment of an existing or novel signaling protocol and, in particular, we do not use a SIP deployment. SIP requires, in fact, the use of trusted proxies and has been documented as leaking metadata to network observers [21,43]. Furthermore, with the exception of the audio codec negotiation, SIP functionalities largely overlap mechanisms already offered by Tor [21,43].

A caller can discover a callee by looking up a specific onion service identifier using the Tor DHT. This onion service identifier is obtained by other means, e.g., by using an anonymous chat service. The identifier can also be public while still preserving anonymity, as Tor prevents an external observer from determining that a specific client opens a circuit to a specific onion service. For instance, journalists could advertise an anonymous onion service for whistleblowers to use. We note that client-side authorization, as defined in the Tor rendezvous specification [54], could enable a callee to only allow calls from a whitelist of callers, but we leave the integration of this functionality to future work.

In the current DONAR implementation, the codec and its configuration are hardcoded. Codec and configuration negotiation require, unlike discovery, only communication between the two parties, and could employ a protocol similar to the subset of SIP dedicated to this task. We also leave this implementation to future work.

5 Security

DONAR leverages Tor without deploying additional infrastructure or modifying Tor itself. As a result, DONAR inherits the security assumptions and shortcomings of Tor. For instance, like Tor, DONAR does not provide protection from adversaries that can control the *entire* network [20,59] to perform traffic-correlation attacks [40,82]. Nevertheless, in terms of guarantees, it is reasonable to wonder whether DONAR worsens the security properties of Tor and to what extent.

In the definition of the so-called predecessor attack, Wright [82] observed that repeatedly creating new circuits causes clients to continuously degrade their security while increasing the probability that they will eventually choose a malicious relay as the first node of a circuit. Wright [81] proposed to address this problem by using what is now known as guards. Specifically, each Tor client chooses a small number of guards and uses them for all the circuits it ever creates. This suggests that DONAR's impact on security depends mainly on the fact that it can use a larger number of guards than the standard Tor implementation. We evaluate this impact from the perspective of three threats: (1) one endpoint deanonymizing the other, (2) an attacker that controls some relays or ASes and that tries to identify DONAR users, and (3) the same attacker deanonymizing both endpoints of a call and finally breaking anonymity.

Deanonymizing the other endpoint. According to the AnoA classification [6], sender/recipient anonymity refers to the ability to hide one endpoint's identity from the other. As discussed by Wright *et al.* [81], in a system with *c* corrupted relay nodes out of *n* and 1 guard per user, the probability of an endpoint's de-anonymizing the other is $\frac{c}{n}$. If we increase the number of guards to *g*, this probability becomes $1 - (1 - \frac{c}{n})^g$, which, for small values of $\frac{c}{n}$, can be approximated from above

by its first-order Taylor/Maclaurin expansion $g\frac{c}{n}$. Like most previous work, this analysis focuses on a random distribution of compromised guards. Adversaries can also leverage path selection algorithms to strategically place malicious guards and increase their probability of being selected although countermeasures exist [78].

Identifying DONAR users. Identifying a DONAR endpoint is equivalent to de-anonymizing any onion service, i.e., identifying which client node is reachable through this service. An adversary controlling a guard relay and knowing the onion address of a callee may observe traffic and employ traffic fingerprinting techniques [10,45,55,61,65] or use a fake DONAR client and perform timing attacks [58] to identify that a specific client is accepting DONAR calls. The use of several (g) guards in DONAR also increases the probability of this attack to $1-(1-\frac{c}{n})^g$, and thus by a factor of g for small values of $\frac{c}{n}$, like for the de-anonymization of one endpoint. This attack can however be mitigated by using the client-authorization feature offered by V3 Onion Services [54]. Finally, while several authors have shown that an adversary could locate onion service endpoints even when they were not publicly advertised [9,45,55,61], they have also proposed solutions to the Tor community.

De-anonymizing an ongoing call. To de-anonymize an ongoing call, an attacker needs to control guard nodes at both endpoints and employ traffic-correlation techniques [40]. As a result, like for the first two threats, the choice of the number of guards used by DONAR identifies a tradeoff between the likelihood of this attack and the performance of a call. In particular, since the attacker needs to control at least one guard on each side of the call, the associated probability grows from $(\frac{c}{n})^2$ with one guard to $(1-(1-\frac{c}{n})^g)^2$ with g guards. This implies that it grows even more slowly for small values of $\frac{c}{n}$ than the two previous probabilities.

Finally, we also observe that passive traffic correlation attacks turn out to be more difficult to perform when multiple calls are ongoing as DONAR's traffic patterns do not vary between different calls. In this case, a passive attack must observe the start and/or the end of a call to be effective.

DONAR security configurations.

As discussed above, increasing the number of guards improves performance but it also increases the attack surface. For this reason, DONAR implements three security configurations that strike different tradeoffs between privacy and performance, as illustrated in Figure 7. We emphasize that each configuration sets up the unmodified Tor client via its legacy API. DONAR systematically uses 12 links, but link settings are different in each configuration. The *Default* configuration provides a security strength similar to the legacy Tor client with default Tor link settings, i.e., each link has 6 relays, and each client employs only 2 guards.[7] The *2-hop* configuration

---

[7]Even though Tor's documentation discusses using only one guard, the default client uses two.
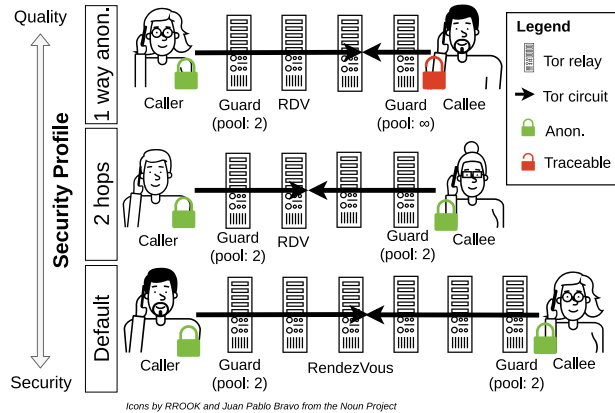
Figure 7: Security configurations.

sets up the Tor client so that each created link has two fewer Tor relays compared to Tor's default link settings. Finally, the *1-way-anonymity* configuration totally removes the anonymity of the callee using a *single* Tor relay (without the guard pool constraint) between the callee and the rendezvous point.

Security Discussion. Each of the threats we identified above relies on the control of at least one guard relay per affected endpoint. As discussed above, DONAR does not modify the guard configuration when providing anonymity for a user. Moreover, the use of guards decorrelates the number of links and the de-anonimization probability: using 12 links at once does not expose a user more than using only one. Additionally, compared to the *Default* configuration, the *2-hop* one reduces the number of relays in links by two. Decreasing the number of relays in links has been long debated in the Tor community. The main rationale for using 3-relay circuits (and thus 6-relay links) is that it makes it more difficult for an adversary that controls the last relay to identify the entry guard. On the other hand, an adversary can overcome this protection with relatively low investment in additional relays, and 3-relay circuits are more vulnerable to attacks based on denial of service [8]. These observations motivate our choice of 2-relay circuits with better latency in our *2-hop* configuration. Finally, the *1-way-anonymity* configuration does not provide anonymity to the callee but does not hamper the anonymity of the caller. Moreover, this mode is a standard feature of the Tor daemon that is used in production (e.g., by Facebook [1]).

Finally, we emphasize that DONAR users may also explore entirely different security configurations, by changing the number of Tor guards and/or relays for links, according to their own expected tradeoffs between performance and security.

6 Evaluation

DONAR is available open-source at `https://github.com/CloudLargeScale-UCLouvain/Donar`. The DONAR proxy interfaces a VoIP application with the Tor client.[8] We use two applications: (1) a configurable RTP emulator allowing a fine-grained control on the frames sent between parties, and running multiple occurrences of an experiment to study statistical variations; and (2) the actual `gstreamer` VoIP application using the OPUS codec. We deploy two isolated instances of either application on the same machine to accurately measure one-way delays for packets sent over Tor.

Tor's performance varies over time, with failures, disconnections, and latency spikes as identified in Section 3. Unless mentioned otherwise, we run each experiment a total of 64 times and present the distribution of results. We run the same configuration over a long time span, typically 5 hours, and also compare different configurations running in parallel.

6.1 Performance & comparison to SOTA

We start with the evaluation of the global performance of DONAR and its ability to meet the requirements summarized in Table 1. We use an audio stream of 32 kbps with a rate of 25 frames per second. We configure DONAR as follows: The window duration is $w = 2s$ and we open a total of $n_{LINKS} = 12$ links including $n_{1ST} = 3$ links, $n_{2ND} = 3$ links, and $n_{INACTIVE} = 6$ links. We present a comprehensive analysis of the influence of these parameters in Section 6.2.

We consider the six possible variants of DONAR using either of the two scheduling policies ALTERNATE and DOUBLE-SEND combined with one of the three security configurations (*Default*, *2 hops*, or *1-way anonymity*). In addition, we implement two approaches representing the state of the art. SIMPLE is the direct use of a single Tor link to transfer VoIP data. It represents our reference in terms of bandwidth usage for the ALTERNATE policy. TORFONE implements the duplication strategy used in TorFone [24]: It sends each new packet on two links, representing a reference for bandwidth usage for the DOUBLE-SEND policy.

No call interruption. We start by studying the percentage of dropped calls for all configurations. We run 96 instances of a 90-minute call for each combination and count the percentage of dropped calls. For SIMPLE, a broken Tor link invariably results in a dropped call. The DONAR variants and TORFONE, instead, re-establish broken links and thus consider their calls dropped whenever they miss 25 consecutive frames. Figure 8 presents the results. All DONAR variants perform better than the previous approaches and meet the goal of less than 2% of dropped calls. We only record, in fact, dropped calls for the most conservative of our setups, i.e., combining the ALTERNATE policy with the *default* configuration, and even then not

---

[8]The Tor software is evolving quickly, especially considering v3 onions. To benefit from latest bug fixes, we compiled Tor from branch `maint-0.4.4` (commit `09a1a34ad1`) and patch #256.
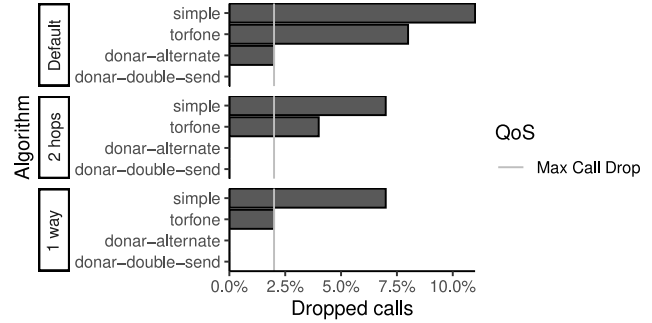
Figure 8: Dropped calls after 90 minutes for SIMPLE, TORFONE, and DONAR setups.
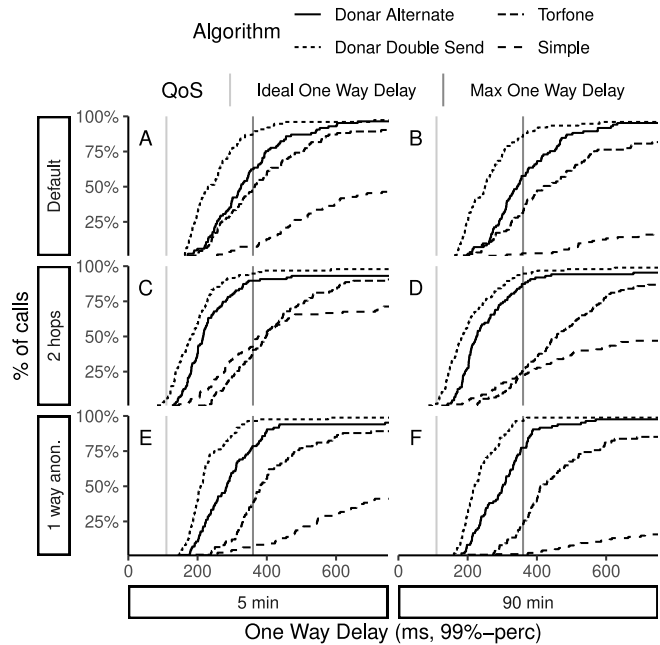
Figure 9: Latency comparison between SIMPLE, TORFONE, DONAR ALTERNATE and DONAR DOUBLE-SEND.

exceeding 2%. TORFONE only meets the goal in the *1-way anonymity* configuration.

Interactive conversations & good voice quality. These objectives require a sufficient bitrate—met by using a 32 kbps bitrate in our experiments—and receiving at least 99% of VoIP frames within the maximum acceptable latency. The OPUS codec can, indeed, mask the loss of 1% of the frames with no perceptible quality degradation.

We present the distributions of frame delivery latencies in Figure 9. Our mouth-to-ear latency objective is 150 ms, and our limit is 400 ms. As T_{frame}=40 ms, we wish network delays for delivering frames to be of 110 to 360 ms. We use two vertical lines to denote these boundaries.

For all security policies and call durations, the DONAR DOUBLE-SEND algorithm provides at least 87% (*Default*, 90 minutes) of successful calls. Considering only our optimized security policies, the ratio of successful calls is even higher at

95%. These results must be compared to TORFONE, as both approaches send the same amount of data on the wire. TOR-FONE enables as low as 23% (*1-way anon.*, 90 minutes) and at most 47% (*Default*, 5 minutes) of successful calls. Compared to DONAR DOUBLE-SEND's worst performance (*Default*, 90-minute configuration), there is a 55-point difference with TORFONE in favor of DONAR.

Conversely, we observe that DONAR ALTERNATE does not fit all configurations: for its *Default* security policy, it enables only 62% (resp. 57%) of successful calls for 5 minutes (resp. 90 minutes). Results are better with *1-way anon.*: 78% (resp. 77%) for 5-minute (resp. 90-minute) calls. However, only the *2-hop* configuration seems to offer acceptable quality, enabling at least 87% of successful calls. Compared to the SIMPLE mode that sends the same amount of data, this is a 55-point gain points compared to DONAR's worst performance. With the *2-hop* configuration, it is a 43 points (resp. 65 points) for 5-minute (resp. 90-minute) calls improvement on SIMPLE.

To conclude, DONAR DOUBLE-SEND is able to offer a high ratio of successful calls in most situations (87%+ compared to 23%+ for TORFONE); it is a versatile solution at the cost of added redundancy on the wire. In comparison, DONAR ALTERNATE has no overhead but is way more sensitive to the configuration: it only works well with the *2-hop* security policy (87%+ compared to 46%+ for SIMPLE). With a difference of at most 4% between the 5-minute and 90-minute measurements, DONAR adds a new interesting property: latency stability over time. We argue that our two sending policies represent a significant improvement in terms of delay compared to the state of the art.

Using the `gstreamer` VoIP client. We experiment with the replay of an audio file using the `gstreamer` VoIP application. We collect statistics about its jitter buffer. `gstreamer` only allows a static-size jitter buffer. We configure this buffer based on our previous experiments, so as to absorb latencies between the minimum observed latency and the 99th-perc. latency, and count the number of calls that systematically meet latency requirements out of the 64 experiments done for each configuration. Our results confirm that DONAR DOUBLE-SEND is able to meet the 360 ms latency threshold for most experiments in all configurations, while the ALTERNATE policy works best under the *2-hop* configuration. We also confirmed empirically the results obtained under the *2-hop* configuration and the two scheduling policies by performing actual calls between two laptops: we could not detect any degradation in sound quality throughout any of the calls.

6.2 Microbenchmarks

In the following, we present an analysis of the influence of each of DONAR's parameters, and of the complementarity of its mechanisms. We focus on the six possible DONAR variants and, to factor out the impact of security configurations, we also consider a version of DONAR using 4 relays per link and an unlimited number of guards.
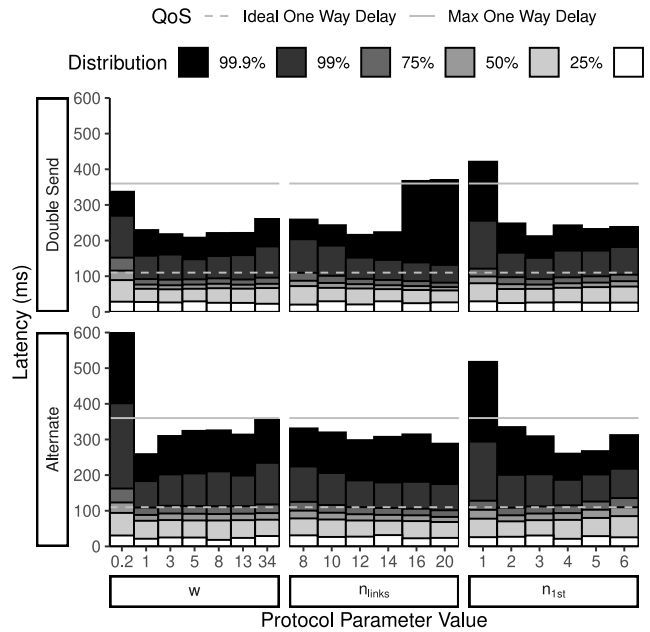
Figure 10: Impact of protocol parameters (w, n_{LINKS} and $n_{\mathrm{1ST}} = n_{\mathrm{2ND}}$) on frame delivery latencies.

Protocol parameters. DONAR has 3 main parameters: w, n_{LINKS}, n_{1ST} (we use $n_{\mathrm{1ST}} = n_{\mathrm{2ND}}$). In the experiments reported in the previous section, we employed the default values of $w = 2s$, $n_{\mathrm{LINKS}} = 12$, and $n_{\mathrm{1ST}} = n_{\mathrm{2ND}} = 3$. We detail in the following how we selected this default configuration.

We present, in Figure 10, an analysis of the influence of each parameter on the distribution of frame delivery latencies. Parameter w determines how far in the past we consider out-of-order metrics when computing link scores. It also determines how many times we need to probe a link before deciding to stop using it. A lower value of w enables a fast reaction at the risk of switching too many links with unreliable scores, while a larger value promotes links that are stable over time. We can observe on the left side of Figure 10 that the best value of w for the DOUBLE-SEND policy is 5s, while the best for ALTERNATE appears to be 2s. Additional benchmarks on the $[1, 8]$ range with a smaller step led us to select the latter value as the default.

The n_{LINKS} parameter controls the total number of open links and, therefore, both the level of achievable diversity and the load of route maintenance on the Tor network. We evaluate n_{LINKS} values from 8 to 20. The ALTERNATE policy performs best with 20 links, while the DOUBLE-SEND policy performs best with 12 links. To limit the load on Tor, we select this latter value as the default.

Finally, parameter $n_{\mathrm{1ST}} = n_{\mathrm{2ND}}$ directly controls the number of links that are actively used to send packets. On the one hand, for a given value of n_{LINKS}, a small value of n_{1ST} increases the likelihood of selecting only good-performing links. On the other hand, a large value increases diversity and the frame rate on each link, resulting in higher stability as we have shown
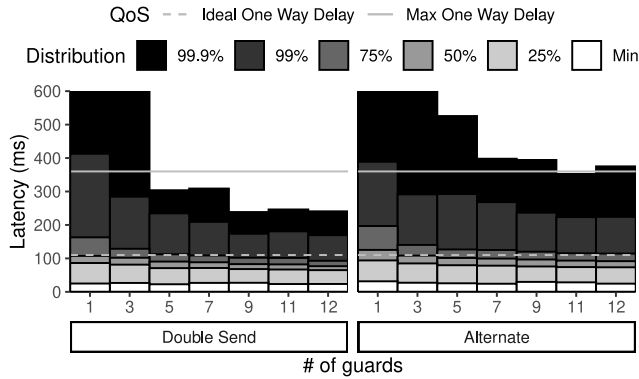
Figure 11: Impact of Tor guards number on latencies.
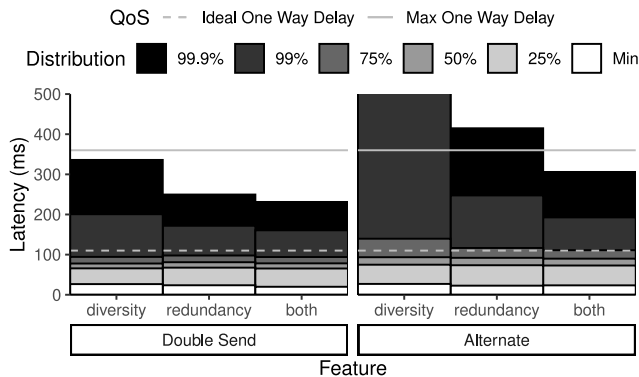
Figure 12: Diversity & redundancy complementarity.

in Section 3. Using $n_{1ST} = 1$ yields high latencies with either variant, while $n_{1ST} = 3$ or $n_{1ST} = 4$ offers a good compromise. We choose $n_{1ST} = 3$ as our default value.

Impact of the size of the guard pool. We considered using different sizes for the guard pool for the different security configurations detailed in Section 5. We further explore the impact of this parameter on DONAR's performance. Our results, shown in Figure 11, confirm that, in order to achieve the best latency, it is preferable to have as many guards as the number of links, in our case $n_{LINKS} = 12$. This number is, however, the result of a compromise with the attack surface. In our performance evaluation, we chose to stay conservative by not modifying the number of guards but we demonstrate here this choice has a cost in terms of performance.

Complementarity of diversity and redundancy. We analyze to which extent the two enabling mechanisms of DONAR, diversity and redundancy, contribute to its performance. We present in Figure 12 latency when using only link selection (diversity), using only redundancy by piggybacking, and using both. Activating both features is clearly beneficial for both scheduling policies, but, unsurprisingly, the impact of redundancy by piggybacking on high percentiles of the distribution is larger for the ALTERNATE strategy than for the DOUBLE-SEND strategy, as the latter enables redundancy by sending packets twice.

We further wish to understand how diversity and redundancy interact when used simultaneously, by analyzing, for each frame, which group of links delivers it for the first time, and whether this first delivery concerns a primary or a duplicate copy. The first delivery of a frame, indeed, results from a *race* between two send operations (with the ALTERNATE policy) and four send operations (with DOUBLE-SEND).

When using the ALTERNATE policy, 94% of the primary frame copies sent on a link of the L_{1ST} group arrive first. In 6% of the cases, the first copy that is received is the duplicate sent 40 ms later over an L_{2ND} link. When primary frame copies are sent over L_{2ND} links, however, only 48% arrive before the duplicate copy sent over an L_{1ST} link; 52% of the frames arrive first as the duplicate copy, despite the latter being sent 40 ms later. When using the DOUBLE-SEND policy, 73% of the frames are received first as a primary copy on the L_{1ST} link, 14% are received as a primary copy on an L_{1ST} link, and only 13% are received as a duplicate copy. Using L_{2ND} links remains useful. It provides more diversity, while still leveraging the reliability of the best links. Moreover, it decreases the load on each individual link, reducing the risk of performance degradation on each of them.

Link monitoring effectiveness. Appendix A presents a supplementary study of the effectiveness of link monitoring, where we analyze a trace of link classification and selection.

7 Related Work

VoIP over anonymization networks poses significant challenges as it combines the need for strong security with low and stable latency requirements. Three main families of anonymization networks have emerged: onion-route-, mix-net- and DC-net-based networks. The former do not protect from global adversaries that control the entire network whereas the latter two do.

The objective of DONAR is to leverage a readily-available system. Only two anonymity networks satisfy this requirement, Tor and Vuvuzela [76]. We discuss, nonetheless, the practicability of VoIP over a larger set of existing approaches, even if they are not effectively deployed.

Onion-route-based networks. Sharma *et al.* [70] called to re-think the feasibility of voice calling over Tor and claim that VoIP is feasible over Tor. However their analysis suffers from several shortcomings: they consider average latency instead of tail latency, they do their measurements only for 30 seconds, they do not evaluate dropped calls, they only provide one-way anonymity, etc. Karopoulos *et al.* [21, 43] explore the porting of SIP infrastructures on Tor. The main principle of their work is to preserve privacy in the SIP signaling protocol, in contrast with DONAR that leverages Tor's built-in mechanisms for establishing calls. The RTP stream is transmitted using a single Tor onion link. This approach behaves like SIMPLE from our experimental evaluation in this respect. TorFone [24] tries to improve latency by duplicating traffic over only two onion

links without any scheduling and monitoring mechanisms. As demonstrated in Section 6, the TORFONE policy is not sufficient to meet VoIP requirements.

Mix-net-based networks. Mix networks [13] batch and shuffle packets via *mix nodes* to prevent attackers from performing global traffic analysis. However, in doing so, they inherently incur high latency, which makes them unusable in latency-sensitive applications. A key solution to reduce packet delivery times consists in using cover traffic to prevent the mixes from having to wait too long before having enough packets to send a batch. Accordingly, the challenge faced by the latest research on mix-nets, such as Karaoke [48], Vuvuzela [76], Riffle [46], Loopix [64], Aqua [51], and Stadium [73], consists in designing an adequate mix-net with the best tradeoff between minimizing the necessary cover traffic while guaranteeing good resilience to traffic analysis. In their best-case usage scenario, these approaches drastically reduce latency from several hundred seconds to a few seconds, but this remains very far from VoIP requirements.

DC-net-based networks. Latency can be reduced by avoiding batching. Instead of using mix nodes, Dining-Cryptographer Networks (DC-nets) rely on anonymous broadcast among all network participants [13]. DC-nets have two inherent shortcomings: (i) they incur a high bandwidth overhead, i.e., the number of messages exchanged to send one message anonymously grows quadratically with the number of network participants, and (ii) they are vulnerable to denial of service attacks from malicious participants that can jam the whole network. Being resistant to such attacks requires, for instance, the use of zero-knowledge proofs to detect misbehavior but this is very costly in computation and results in increased delivery latency [25]. Consequently, a number of research works on DC-nets have emerged in recent years. Dissent [16, 80], Riposte [15], and Verdict [17] resist jamming attacks while trying to provide the best tradeoff between reducing the number of exchanged messages (e.g by splitting the network into smaller parts) and the impact of computational cost on latency. However, despite their efforts, their latency remains far too high for VoIP and increases with the number of users.

Anonymization networks designed for VoIP. Herd [50] is based on the mix-net principle. It was specifically designed for VoIP. Its hybrid approach uses mix nodes along with super peers organized in trust zones. Herd can provide VoIP on its anonymity network with good resistance against global adversaries. Its evaluation shows expected latency values of 400 ms. The recent work on Yodel [49] removes the concept of trust zones and supports higher percentages of dishonest nodes than Herd. However, this comes at the cost of latency increasing with the probability of having dishonest mix nodes. For instance, in a Tor-like environment (i.e., $\sim 20\%$ of malicious servers) latency already reaches ~ 900 ms. To counterbalance this latency, Yodel uses a codec with poorer quality than OPUS. Even if both Herd and Yodel are promising designs, neither is currently deployed. Today's whistleblowers are, therefore, unable to communicate using these systems. Moreover, we point out that the evaluation of both systems has been performed in optimal conditions, and their performance in settings comparable to Tor's deployment remains unstudied. For instance, Yodel is evaluated on 100 powerful Amazon EC2 servers with no external interference. DONAR, on the other hand, satisfies VoIP latency requirements, even if Tor constantly relays traffic generated by over 2 million daily users. To summarize, Tor and Vuvuzela represent the only anonymization networks that are readily available and widely deployed today. Since Vuvuzela cannot support VoIP due to its high latency, DONAR over Tor represents the only solution that enables privacy-conscious users to communicate anonymously using VoIP and with a good QoE.

Latency improvements on Tor. We also reviewed existing proposals to improve latency in Tor. This latency depends on two main factors: (i) queuing delays (time spent in a relay), and (ii) transmission delays (time spent on the "wire", between two Tor relays). Ting [11] and LASTor [2] both reduce transmission delays by modifying the path selection algorithm. However, latency spikes are due to queuing delays [19], particularly because Tor does not perform any centralized load balancing of traffic. To reduce queuing delays, improved traffic scheduling policies have been integrated in the latests versions of Tor [38, 39], but we still observe latency spikes. Alternative path selection algorithms use historical data on relay performance [71, 79] or probe circuits upon their creation [5]. They are inefficient for VoIP as latency spikes are ephemeral; predictions are outdated after a few seconds.

Multipath. We are not the first to advocate for multipath. MORE [47] proposes to route independently each cell, but it is not designed to be used with circuits like in Tor. MPTCP [22, 23, 29] aggregates TCP links over multiple network interfaces. However, it makes assumptions (e.g., latency is independent of traffic) that do not hold over Tor. In response, dedicated multipath protocols specially tailored for onion routing networks [3, 18, 42, 83] emerged. Nevertheless, compared to DONAR, none of these approaches optimize tail latency as required by all real-time protocols, including VoIP.

8 Conclusion

We presented DONAR, a solution for readily-available, anonymous, and high-quality VoIP calls using the challenging but existing Tor network. DONAR circumvents Tor's inability to support the networking requirements of VoIP by sending audio frames over a diversity of links and using redundancy. It offers different tradeoffs between performance and security and successfully enables high-quality VoIP calls, e.g., with latency below 360ms during an entire 90-minute call.

References

[1] Tor project issue trackers: Facebook's onion site is a single hop onion, but clicking on the Tor onion icon shows that it is a 6 hop circuit (issue #23875). https://gitlab.torproject.org/legacy/trac/-/issues/23875.

[2] Masoud Akhoondi, Curtis Yu, and Harsha V Madhyastha. LASTor: A low-latency AS-aware Tor client. In *IEEE Symposium on Security and Privacy*, S&P, 2012.

[3] Mashael AlSabah, Kevin Bauer, Tariq Elahi, and Ian Goldberg. The path less travelled: Overcoming Tor's bottlenecks with traffic splitting. In *International Symposium on Privacy Enhancing Technologies*, PETS. Springer, 2013.

[4] Soren Vang Andersen, Alan Duric, Henrik Astrom, Roar Hagen, W. Bastiaan Kleijn, and Jan Linden. Internet Low Bit Rate Codec (iLBC). Request for Comments (RFC) 3951, Internet Engineering Task Force (IETF), December 2004.

[5] Robert Annessi and Martin Schmiedecker. Navigator: Finding faster paths to anonymity. In *European Symposium on Security and Privacy*, EuroS&P. IEEE, 2016.

[6] Michael Backes, Aniket Kate, Praveen Manoharan, Sebastian Meiser, and Esfandiar Mohammadi. AnoA: A framework for analyzing anonymous communication protocols. In *26th Computer Security Foundations Symposium*, CSF. IEEE, 2013.

[7] Luca Barbato. RTP payload format for Vorbis encoded audio. Request for Comments (RFC) 5215, Internet Engineering Task Force (IETF), August 2008.

[8] Kevin Bauer, Joshua Juen, Nikita Borisov, Dirk Grunwald, Douglas Sicker, and Damon McCoy. On the optimal path length for Tor. In *3rd Hot Topics in Privacy Enhancing Technologie*, HotPets, 2010.

[9] Alex Biryukov, Ivan Pustogarov, and Ralf-Philipp Weinmann. Trawling for tor hidden services: Detection, measurement, deanonymization. In *Symposium on Security and Privacy*, S&P. IEEE, 2013.

[10] Xiang Cai, Xin Cheng Zhang, Brijesh Joshi, and Rob Johnson. Touching from a distance: Website fingerprinting attacks and defenses. In *ACM conference on Computer and communications security*, CCS, 2012.

[11] Frank Cangialosi, Dave Levin, and Neil Spring. Ting: Measuring and exploiting latencies between all tor nodes. In *Internet Measurement Conference*, IMC, 2015.

[12] David L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2), 1981.

[13] David L. Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of cryptology*, 1(1), 1988.

[14] Opus Codec. Codec landscape. https://opus-codec.org/comparison/, 2020.

[15] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. Riposte: An anonymous messaging system handling millions of users. In *Symposium on Security and Privacy*, S&P. IEEE, 2015.

[16] Henry Corrigan-Gibbs and Bryan Ford. Dissent: accountable anonymous group messaging. In *17th ACM conference on Computer and communications security*, CCS, 2010.

[17] Henry Corrigan-Gibbs, David Isaac Wolinsky, and Bryan Ford. Proactively accountable anonymous messaging in Verdict. In *22nd USENIX Security Symposium*, 2013.

[18] Wladimir De la Cadena, Daniel Kaiser, Asya Mitseva, Andriy Panchenko, and Thomas Engel. Analysis of multi-path onion routing-based anonymization networks. In *IFIP Annual Conference on Data and Applications Security and Privacy*, DBSec. Springer, 2019.

[19] Prithula Dhungel, Moritz Steiner, Ivinko Rimac, Volker Hilt, and Keith W Ross. Waiting for anonymity: Understanding delays in the Tor overlay. In *10th International Conference on Peer-to-Peer Computing*, P2P. IEEE, 2010.

[20] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. Technical report, Naval Research Lab Washington DC, 2004.

[21] Alexandros Fakis, Georgios Karopoulos, and Georgios Kambourakis. OnionSIP: Preserving privacy in SIP with onion routing. *J. Univers. Comput. Sci.*, 23(10), 2017.

[22] Alexander Froemmgen, Jens Heuschkel, and Boris Koldehofe. Multipath TCP scheduling for thin streams: Active probing and one-way delay-awareness. In *International Conference on Communications*, ICC. IEEE, 2018.

[23] Alexander Frommgen, Tobias Erbshäußer, Alejandro Buchmann, Torsten Zimmermann, and Klaus Wehrle. ReMP TCP: Low latency multipath TCP. In *International Conference on Communications*, ICC. IEEE, 2016.

[24] Van Gegel. TORFone: secure VoIP tool. http://torfone.org/, 2013.

[25] Philippe Golle and Ari Juels. Dining cryptographers revisited. In *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2004.

[26] GStreamer. Gstreamer: open source multimedia framework. https://gstreamer.freedesktop.org/, 2020.

[27] Ben Doherty (The Guardian). Vodafone australia admits hacking fairfax journalist's phone. https://www.theguardian.com/business/2015/sep/13/vodafone-australia-admits-hacking-fairfax-journalists-phone, 2015.

[28] Yi Han, Damien Magoni, Patrick Mcdonagh, and Liam Murphy. Determination of bit-rate adaptation thresholds for the opus codec for voip services. In *Symposium on Computers and Communications*, ISCC. IEEE, 2014.

[29] Mark Handley, Olivier Bonaventure, Costin Raiciu, and Alan Ford. TCP extensions for multipath operation with multiple addresses. Request for Comments (RFC) 6824, Internet Engineering Task Force (IETF), January 2013.

[30] G. Herlein, J. Valin, A. Heggestad, and A. Moizard. RTP payload format for the Speex codec. Request for Comments (RFC) 5574, Internet Engineering Task Force (IETF), June 2009.

[31] Stephan Heuser, Bradley Reaves, Praveen Kumar Pendyala, Henry Carter, Alexandra Dmitrienko, William Enck, Negar Kiyavash, Ahmad-Reza Sadeghi, and Patrick Traynor. Phonion: Practical protection of metadata in telephony networks. *Proceedings on Privacy Enhancing Technologies*, 2017(1), 2017.

[32] Christian Hoene, Jean-Marc Valin, Koen Vos, and Jan Skoglund. Summary of Opus listening test results. https://tools.ietf.org/html/draft-ietf-codec-results-03, 2013.

[33] Jan Holub, Michael Wallbaum, Noah Smith, and Hakob Avetisyan. Analysis of the dependency of call duration on the quality of VoIP calls. *IEEE Wireless Communications Letters*, 7(4):638–641, 2018.

[34] Monty Icenogle. T-mobile does have a hard 4 hour single call duration limit. https://kd6cae.livejournal.com/271120.html, 2015.

[35] ITU. ITU-T recommendation G.114, "one way transmission time". https://www.itu.int/rec/T-REC-G.114, 2003.

[36] ITU. E.800 : Definitions of terms related to quality of service, 2008.

[37] ITU. G.1028: End-to-end quality of service for voice over 4G mobile networks. https://www.itu.int/rec/T-REC-G.1028, 2019.

[38] Rob Jansen, John Geddes, Chris Wacek, Micah Sherr, and Paul Syverson. Never been KIST: Tor's congestion management blossoms with kernel-informed socket transport. In *23rd USENIX Security Symposium*, 2014.

[39] Rob Jansen, Matthew Traudt, John Geddes, Chris Wacek, Micah Sherr, and Paul Syverson. Kist: Kernel-informed socket transport for Tor. *ACM Transactions on Privacy and Security (TOPS)*, 22(1):1–37, 2018.

[40] Aaron Johnson, Chris Wacek, Rob Jansen, Micah Sherr, and Paul Syverson. Users get routed: Traffic correlation on Tor by realistic adversaries. In *ACM SIGSAC conference on Computer & communications security*, CCS, 2013.

[41] kamedo2. Results of the public multiformat listening test. https://listening-test.coresv.net/results.htm, 2014.

[42] Hasan T Karaoglu, Mehmet Burak Akgun, Mehmet Hadi Gunes, and Murat Yuksel. Multi path considerations for anonymized routing: Challenges and opportunities. In *5th International Conference on New Technologies, Mobility and Security*, NTMS. IEEE, 2012.

[43] Georgios Karopoulos, Alexandros Fakis, and Georgios Kambourakis. Complete SIP message obfuscation: PrivaSIP over Tor. In *9th International Conference on Availability, Reliability and Security*. IEEE, 2014.

[44] Byeong Hoon Kim, Hyoung-Gook Kim, Jichai Jeong, and Jin Young Kim. VoIP receiver-based adaptive playout scheduling and packet loss concealment technique. *IEEE Transactions on consumer Electronics*, 59(1):250–258, 2013.

[45] Albert Kwon, Mashael AlSabah, David Lazar, Marc Dacier, and Srinivas Devadas. Circuit fingerprinting attacks: Passive deanonymization of Tor hidden services. In *24th USENIX Security Symposium*, 2015.

[46] Albert Kwon, David Lazar, Srinivas Devadas, and Bryan Ford. Riffle: An efficient communication system with strong anonymity. *Proceedings on Privacy Enhancing Technologies*, 2016(2):115–134, 2016.

[47] Olaf Landsiedel, Alexis Pimenidis, Klaus Wehrle, Heiko Niedermayer, and Georg Carle. Dynamic multipath

onion routing in anonymous peer-to-peer overlay networks. In *IEEE Global Telecommunications Conference*, GlobeCom, 2007.

[48] David Lazar, Yossi Gilad, and Nickolai Zeldovich. Karaoke: Distributed private messaging immune to passive traffic analysis. In *13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2018.

[49] David Lazar, Yossi Gilad, and Nickolai Zeldovich. Yodel: strong metadata security for voice calls. In *27th ACM Symposium on Operating Systems Principles*, SOSP, 2019.

[50] Stevens Le Blond, David Choffnes, William Caldwell, Peter Druschel, and Nicholas Merritt. Herd: A scalable, traffic analysis resistant anonymity network for voip systems. In *ACM Conference on Special Interest Group on Data Communication*, SIGCOMM, 2015.

[51] Stevens Le Blond, David Choffnes, Wenxuan Zhou, Peter Druschel, Hitesh Ballani, and Paul Francis. Towards efficient traffic-analysis resistant anonymity networks. *ACM SIGCOMM Computer Communication Review*, 43(4):303–314, 2013.

[52] Yi J Liang, Nikolaus Farber, and Bernd Girod. Adaptive playout scheduling and loss concealment for voice communication over IP networks. *IEEE Transactions on Multimedia*, 5(4):532–543, 2003.

[53] Zhen Ling, Junzhou Luo, Wei Yu, and Xinwen Fu. Equal-sized cells mean equal-sized packets in Tor? In *International Conference on Communications*, ICC. IEEE, 2011.

[54] Nick Mathewson, George Kadianakis, David Goulet, Tim Wilson-Brown, Hans-Christoph Steiner, Filipo Valsorda, and Roger Dingledine. Tor rendezvous specification - version 3, 2017.

[55] Prateek Mittal, Ahmed Khurshid, Joshua Juen, Matthew Caesar, and Nikita Borisov. Stealthy traffic analysis of low-latency anonymous communication using throughput fingerprinting. In *18th ACM conference on Computer and communications security*, CCS, 2011.

[56] Nick Montfort, Arthur Edelstein, Robert Ransom, and Yawning Angel. Tor project feature tracker: Closed enhancement, "UDP over Tor". https://trac.torproject.org/projects/tor/ticket/7830, 2013.

[57] Sue B Moon, Jim Kurose, and Don Towsley. Packet audio playout delay adjustment: performance bounds and algorithms. *Multimedia systems*, 6(1):17–28, 1998.

[58] Steven J Murdoch and George Danezis. Low-cost traffic analysis of Tor. In *Symposium on Security and Privacy*, S&P. IEEE, 2005.

[59] Steven J Murdoch, Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router (2013 DRAFT v1). https://gitweb.torproject.org/tor-design-2012.git/, 2014.

[60] David Kaplan (Newsweek). Suspicions and spies in silicon valley. https://www.newsweek.com/suspicions-and-spies-silicon-valley-109827, 2006.

[61] Andriy Panchenko, Asya Mitseva, Martin Henze, Fabian Lanze, Klaus Wehrle, and Thomas Engel. Analysis of fingerprinting techniques for Tor hidden services. In *Workshop on Privacy in the Electronic Society*, 2017.

[62] Mike Perry. The move to two guard nodes. https://gitweb.torproject.org/user/mikeperry/torspec.git/tree/proposals/xxx-two-guard-nodes.txt?h=twoguards, 2018.

[63] Mike Perry. Tor's open research topics: 2018 edition | tor blog. https://blog.torproject.org/tors-open-research-topics-2018-edition, 2018.

[64] Ania M Piotrowska, Jamie Hayes, Tariq Elahi, Sebastian Meiser, and George Danezis. The Loopix anonymity system. In *26th USENIX Security Symposium*, 2017.

[65] Tobias Pulls and Rasmus Dahlberg. Website fingerprinting with website oracles. *Proceedings on Privacy Enhancing Technologies*, 2020(1):235–255, 2020.

[66] Maimun Rizal. *A Study of VoIP performance in anonymous network-The onion routing (Tor)*. PhD thesis, Niedersächsische Staats-und Universitätsbibliothek Göttingen, 2014.

[67] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. Request for Comments (RFC) 3261, Internet Engineering Task Force (IETF), June 2002.

[68] Katrin Schoenenberg, Alexander Raake, Sebastian Egger, and Raimund Schatz. On interaction behaviour in telephone conversations under transmission delay. *Speech Communication*, 63:1–14, 2014.

[69] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. Request for Comments (RFC) 3550, Internet Engineering Task Force (IETF), July 2003.

[70] Piyush Kumar Sharma, Shashwat Chaudhary, Nikhil Hassija, Mukulika Maity, and Sambuddho Chakravarty. The road not taken: re-thinking the feasibility of voice calling over Tor. *Proceedings on Privacy Enhancing Technologies*, 2020(4):69–88, 2020.

[71] Robin Snader and Nikita Borisov. A tune-up for Tor: Improving security and performance in the Tor network. In *16th Annual Network & Distributed System Security Symposium*, NDSS, 2008.

[72] Tim Terriberry and Koen Vos. Definition of the Opus audio codec, 2012.

[73] Nirvan Tyagi, Yossi Gilad, Derek Leung, Matei Zaharia, and Nickolai Zeldovich. Stadium: A distributed metadata-private messaging system. In *26th Symposium on Operating Systems Principles*, SOSP. ACM, 2017.

[74] JM. Valin and K. Vos. Updates to the Opus Audio Codec. RFC 8251, October 2017.

[75] JM. Valin, K. Vos, and T. Terriberry. Definition of the Opus Audio Codec. Request for Comments (RFC) 6716, September 2012.

[76] Jelle Van Den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *25th Symposium on Operating Systems Principles*, SOSP, 2015.

[77] Voyced. Is there a maximum call length or duration. https://www.voyced.eu/clients/index.php/knowledgebase/397/Is-there-a-maximum-Call-length-or-duration.html, 2019.

[78] Gerry Wan, Aaron Johnson, Ryan Wails, Sameer Wagh, and Prateek Mittal. Guard placement attacks on path selection algorithms for Tor. *Proceedings on Privacy Enhancing Technologies*, 2019(4):272–291, 2019.

[79] Tao Wang, Kevin Bauer, Clara Forero, and Ian Goldberg. Congestion-aware path selection for tor. In *International Conference on Financial Cryptography and Data Security*, FC. Springer, 2012.

[80] David Isaac Wolinsky, Henry Corrigan-Gibbs, Bryan Ford, and Aaron Johnson. Dissent in numbers: Making strong anonymity scale. In *10th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2012.

[81] Matthew Wright, Micah Adler, Brian N. Levine, and Clay Shields. Defending anonymous communications against passive logging attacks. In *IEEE Symposium on Security and Privacy*, S&P, page 28, USA, 2003. IEEE Computer Society.

[82] Matthew K Wright, Micah Adler, Brian Neil Levine, and Clay Shields. The predecessor attack: An analysis of a threat to anonymous communications systems. *ACM Transactions on Information and System Security*, 7(4):489–522, 2004.

[83] Lei Yang and Fengjun Li. mtor: A multipath Tor routing beyond bandwidth throttling. In *2015 IEEE Conference on Communications and Network Security*, CNS. IEEE, 2015.

A Appendix: Link monitoring effectiveness

We provide in this appendix a supplementary study of the effectiveness of link monitoring, dynamic link classification, and link selection. In particular, we assess whether link classification and selection reflect the behaviors discussed in Section 3.

We start by observing the distribution, over 64 calls, of the number of links that were classified as L_{1ST} *at least* once through the duration of a 90-minute call. This distribution is depicted in Figure 13. Note that we do not consider the first 40 seconds of each call, as DONAR has to bootstrap the process with random scores, and poorly-performing links could be assigned to the L_{1ST} group during this bootstrap. Between 6 and 12 links per call have been considered at least once in the L_{1ST} group in every call, with a majority of 8 to 10 links selected. This confirms our analysis that there is no single link that is consistently performing well in Tor, and that link performance varies significantly over time: Links that are poorly performing at a given time may be the best ones a few minutes later.

We study, in finer detail, the stability of links over time, focusing on a single call using the ALTERNATE policy with the Default configuration. We represent the latency of the first delivery of each frame in the first plot of Figure 14. This is the latency that is observed by the VoIP application. Latency remains low throughout the call. In the second plot, we decompose the latency of frames received on the L_{1ST} and L_{2ND} groups, including the first and second receptions. We can clearly see that the latency of the links in the L_{1ST} group is generally lower, and that outlier values are compensated by lower latency on a link in the L_{2ND} group. The third plot represents the assignment of the 12 links to link groups over
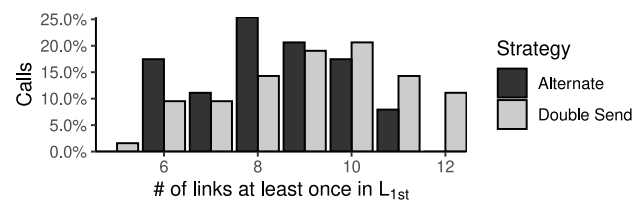
Figure 13: How many links were L_{1ST} at least once?
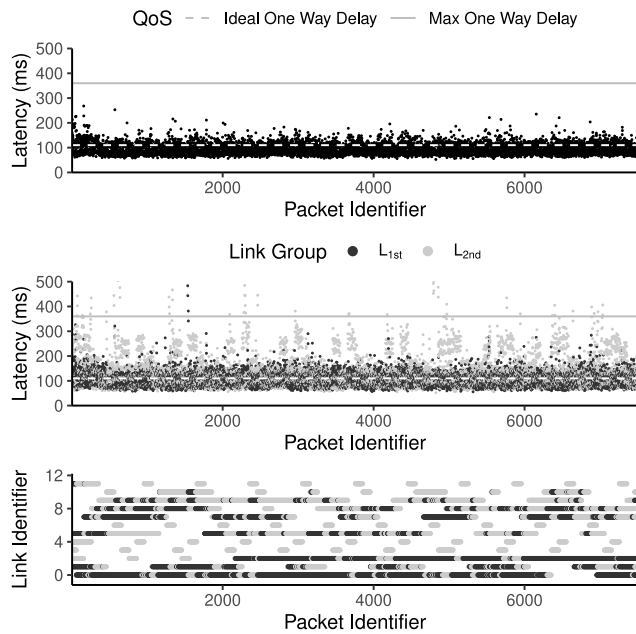
Figure 14: Stability over time.

time. We note that there was no link failure (and therefore no link replacement) in this experiment. Link 0 is, for instance, classified in L_{1ST} for a large part of the call, but suffers a latency spike around frame 6,500 and is rapidly classified in the $L_{INACTIVE}$ group. Link 2, initially in $L_{INACTIVE}$, is promoted 3 times with no effect to the L_{2ND} group, before being selected as L_{1ST} after its fourth promotion. Links 1, 5, 7 and 8 have highly heterogeneous behaviors, while links 3, 4, 6, 11 and 12 have consistently bad behaviors, and only appear in the L_{2ND} group upon their promotion before being quickly deactivated. While these links could be proactively replaced by opening new links, we do not deem it necessary and choose not to impose further link setup load on the Tor network.

Closed-loop Network Performance Monitoring and Diagnosis with SpiderMon

Weitao Wang, Xinyu Crystal Wu, Praveen Tammana[†], Ang Chen, T. S. Eugene Ng
Rice University   [†]Indian Institute of Technology Hyderabad

Abstract

Performance monitoring and diagnosis are essential for data centers. The emergence of programmable switches has led to the development of a slew of monitoring systems, but most of them do not explicitly target posterior diagnosis. On one hand, "query-driven" monitoring systems must be pre-configured with a static query, but it is difficult to achieve high coverage because the right query for posterior diagnosis may not be known in advance. On the other hand, "blanket" monitoring systems have high coverage as they always collect telemetry data from all switches, but they collect excessive data. SpiderMon is a system that co-designs monitoring and posterior diagnosis in a closed loop to achieve low overhead and high coverage simultaneously, by leveraging "wait-for" relations to guide its operations. We evaluate SpiderMon in both Tofino hardware and BMv2 software switches and show that SpiderMon diagnoses performance problems accurately and quickly with low overhead.

1 Introduction

An efficient network monitoring and diagnosis system are essential to meeting the performance requirements of modern applications. Since production clouds have stringent SLAs, even a small network performance degradation may lead to significant application slowdown [13, 30]. Many network performance problems, such as high end-to-end latency, low throughput, and packet drops [38], can be attributed to traffic contention of some kind [4], although across scenarios, the root causes for the contention are diverse (e.g., bursty UDP traffic, ECMP load imbalance, and routing loops).

The emergence of programmable switches has led to a slew of monitoring systems being developed [12, 16, 32, 33, 39, 44, 48], but most of them do not explicitly target posterior diagnosis. For instance, "query-driven" monitoring systems [16, 32] need to be pre-configured with a static query. Since root causes for performance degradation could vary, and there may be a wide variety of reasons for performance problems, it is challenging to select the right query in advance. In principle, one could adaptively change the monitoring query based on the observed symptom; but in practice, many transient problems happen at fine timescales and their sporadic nature

requires always-on monitoring. On the other hand, "blanket" monitoring systems always monitor and collect telemetry data from the switches to achieve high coverage [10, 14, 22, 26, 27]. However, this would result in excessive data that may not be needed by the diagnosis in the first place.

Therefore, having a monitoring and diagnosis system that achieves either low overhead or high coverage is not hard, but achieving both simultaneously is challenging. The key question we explore is whether it is possible to design a streamlined system that performs efficient monitoring but achieves high coverage, achieving the "best of both worlds". We present SpiderMon, a system where the monitoring and diagnosis operations are explicitly designed to work with each other in a closed loop. It enables a suitable tradeoff between accuracy and overhead when debugging network-wide performance problems. To achieve efficient and accurate monitoring, SpiderMon leverages a concept called "wait-for" [46] relations. Since many performance problems stem from in-network contention, "wait-for" relations target such behaviors in the telemetry collection in a precise manner. Moreover, such information is also exactly what is needed in diagnosis. For instance, a victim flow with high latency may have "waited for" many interfering events across multiple hops. By capturing and analyzing such relations, SpiderMon can achieve an effective diagnosis, with precise, targeted, but also high-coverage operations.

Since the symptom of "wait-for" events is usually high latency, SpiderMon uses timing information to trigger reactive telemetry collection. Precisely, SpiderMon detects performance problems when it encounters flows with excessively high queuing delay. After a problem is detected, SpiderMon uses the wait-for relations to track and collect other relevant information in the data plane across the network. For diagnosis, SpiderMon also identifies the root causes of the performance problem by summarizing the most significant wait-for relations from the collected telemetry data. It does so by jointly analyzing wait-for patterns together with other types of network knowledge (e.g., topology) and telemetry data (e.g., flow-level results). In this way, SpiderMon collects telemetry data only when the diagnosis process needs to analyze a problem, and it performs targeted collection based on what

the diagnosis process would require.

To realize this idea, SpiderMon addresses three technical challenges. The first challenge is to detect performance degradation without interfering with actual packet processing. SpiderMon leverages programmable switches to record telemetry data about network traffic. It piggybacks telemetry data in packet headers and checks for performance anomalies. The second challenge is to precisely collect the relevant telemetry information across the network. Relying on wait-for relations, SpiderMon notifies relevant switches and activates telemetry data collection from these locations. Finally, SpiderMon identifies the root causes of the performance problem using the telemetry information and the knowledge of the network configuration. The wait-for relation again is critical for identifying abnormal network behaviors, and for matching those behaviors to the signatures of root causes.

Contributions. Overall, SpiderMon is a *closed-loop* system for monitoring and diagnosing performance problems in the network. We have implemented a prototype of SpiderMon, and our results show that SpiderMon can diagnose performance problems accurately and quickly with low overhead.

2 Motivation

SpiderMon focuses on network performance problems that arise due to contention, which are challenging for at least three reasons. First, network contention may occur due to many root causes, so its diagnosis requires a general mechanism. Second, the root cause can be unpredictable both spatially and temporally, requiring agile solutions that can capture transient problems. A third practical challenge is that the solution must have a sufficiently low overhead on the network. SpiderMon does not target problems that happen because of silent packet drops, packet corruptions, control plane misconfigurations, slow servers, or other causes unrelated to network contention, although it can be used in combination with other techniques for these scenarios.

2.1 Root Causes Are Diverse

To illustrate the diversity of root causes of network performance problems, consider some examples in a 3-layer Clos network as shown in Figure 1.

Micro-bursts. Recent studies [10, 22, 45] found micro-bursts—i.e. momentary surges in traffic volume—to be a common root cause for sporadic excessive delays and packet losses. Detecting and diagnosing a micro-burst requires switch queuing delays to be monitored and the main contributor to queuing delays to be identified before the micro-burst disappears.

Multiple flow contentions. A victim flow encounters multiple contentions at different switches—flow 1 (e.g., a bursty UDP flow) and flow 2 (e.g., a high-priority flow) contend with the victim flow at switch 0 and switch 6, respectively (Figure 1(a)). The end-to-end latency for the victim flow becomes very high. For detection, we need to monitor per-flow
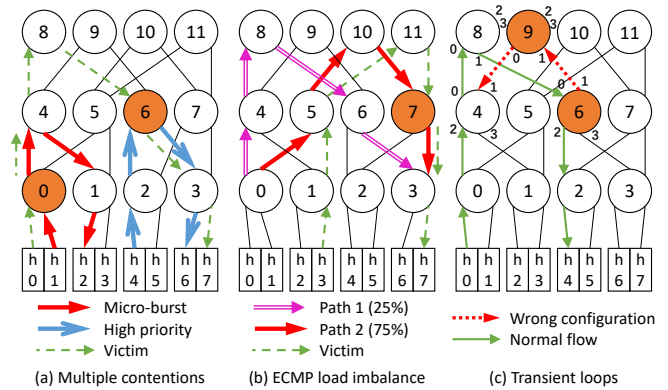
Figure 1: Several performance degradation problems

latency; for diagnosis, information about all contending flows is needed to identify the root causes.

ECMP load imbalance. Due to the skewed nature of flow distributions or imperfect hash mechanisms, ECMP load imbalance is a common problem in data centers [3]. Consider the network in Figure 1(b), where all links are 40Gbps. Switch 0 assigns 25% of the total traffic (32Gbps) to path 1 and 75% to path 2. The victim flow contends with the flows on path 2, which leads to high congestion at switch 7. This could be avoided if switch 0 assigns the traffic for the two paths equally. The root cause for this problem is the imbalanced assignment at switch 0, but the performance degradation occurs at switch 7, which is 3 hops away from switch 0. Once high latency is detected at switch 7, the previous hops' information of the flows involved in the congestion is required for debugging.

Transient/persistent loops. During network updates, the configurations of different switches may not be synchronized. Some switches may fail to execute the reconfiguration commands silently. Under those circumstances, a forwarding loop may form [28]. An example is shown in Figure 1(c), where switches 6 and 9 are wrongly configured, which causes some flows to be stuck in a loop, leading to congestion and packet drops. The incompatible switch configurations should be blamed for the loop in the network. However, to identify the switches that need to be reconfigured, information from all the switches along the loop, namely, switches 6, 9, 4, and 8, needs to be collected for analysis.

2.2 Root Causes Are Unpredictable

There are three key features that make network performance problems challenging to detect or diagnose.

Sporadic. Performance degradation is usually sporadic, occurring occasionally at different places and at an unpredictable time [1]. Any flow may be affected, so detection algorithms need to monitor every flow all the time.

Network-wide. The root causes may be network-wide, e.g., contention at different hops. The interfering flows may even have normal performance [38], despite the fact that they cause performance degradation to other flows. Thus root cause diagnosis requires network-wide monitoring.

Transient. Traffic contentions sometimes are transient and disappear quickly [21]. For instance, transient loops may only form for a short time during network updates, but the performance problem introduced by packet drops may need a much longer time to fully recover. This feature requires the debugging system to maintain fine-grained information about recent events, in case the problems disappear quickly but happen in the network frequently.

2.3 Existing Solutions Fall Short

Existing solutions all fall short in monitoring and diagnosing network performance problems due to the above challenges.

Host-based solutions. Solutions like Trumpet [31] and Dapper [14] rely on end hosts to store telemetry data for diagnosis. But they all use inference algorithms to reconstruct what may have happened in the network from the collected data, which may not be accurate. Instead, SpiderMon collects data from the switches to achieve a better in-network view for diagnosis.

In-network solutions. Some existing solutions also collect telemetry data from the switches. **(i) Blanket telemetry systems** like NetSight [17] and PINT [8] collect information network-wide indiscriminately, even on network nodes unrelated to the problem. Those systems usually have high overheads, and much of the collected data is unnecessary for diagnosis. **(ii) Query-based systems** deploy queries into switches for data collection, such as Sonata [16], Marple [32], FlowRadar [26], and NetSeer [47]. They require that the operators know the nature and location of the problems, but problems could arise from sporadic congestion at random locations. Although in principle, queries can be changed based on the monitoring results, this happens at coarse timescales and cannot capture transient problems. SpiderMon can cover problems that cannot be succinctly defined using static queries and only capture events relevant to the problems.

3 SpiderMon Design

SpiderMon monitors and diagnoses performance problems caused by in-network contention in three steps: 1) SpiderMon encodes every packet's accumulated latency in header fields, and triggers telemetry collection once excessive latency is detected (§3.1); 2) the switch that detects high latency initiates "spider" packets and rapidly delivers them to relevant switches using the wait-for relations; relevant switches receiving spider packets report their telemetry data (§3.2); 3) the root cause analyzer constructs wait-for relations from the evidence for root cause analysis (§3.3).

3.1 Problem Monitoring

Goal: Detect excessive cumulative queuing delays. Rather than wait for the occurrence of harmful events (e.g., packet loss, TCP congestion window back-off), SpiderMon detects the performance problems based on a much earlier sign—abnormal cumulative queuing delays experienced by packets. It reacts quickly to performance degradation.

Design: 1) Use cumulative latency for detection. Instead of storing per-hop latency information in the header, SpiderMon uses cumulative latency to guarantee that the header length stays constant regardless of hop count. The cumulative latency L is updated at every hop based on the current queuing delay and the cumulative latency experienced by the packet so far, $L = L + queuing\_delay$. Every switch on the path checks whether the cumulative delay exceeds the latency threshold. To further reduce overhead, SpiderMon can compress the additional fields to less than 2 bytes by extracting the most significant bits (more in §C.2). **2) Assign different latency thresholds for different traffic types.** Given that the tolerable latency varies for different applications, SpiderMon allows network operators to customize the latency thresholds for different applications. **3) Detect problems and trigger telemetry in the switch data plane.** Unlike some monitoring systems using a central controller to monitor network problems [6, 31, 48], SpiderMon triggers fast reactions in the data plane. The communication delay within the data plane (tens of ns) is much lower than that between the data plane and the control plane (hundreds of μs). **4) Monitor every packet at every hop for target flows.** Compared to sampling-based detection [2, 34], SpiderMon achieves full coverage without losing any important information. Also, rather than detecting problems at the end hosts [9, 24], SpiderMon detects performance problems inside the network and reacts more quickly to the problem. **5) Be transparent to end-hosts.** The latency threshold and cumulative latency are added at the edge switches when packets enter the network and removed when packets leave the network. Hosts remain unchanged.

Consider Figure 1(a) as an example. The victim flow suffers from queuing delay at switches 0 and 6, but the cumulative latency exceeds the threshold only at switch 6. Thus the problem is detected at switch 6, and switch 6 triggers the telemetry collection procedure.

3.2 Telemetry Collection

Goal: Only collect evidence relevant to root cause analysis. SpiderMon maintains a small amount of telemetry information as evidence on the switches to facilitate subsequent diagnosis; this information is not collected from the switches unless needed. First, to minimize the amount of telemetry data collected to the analyzer while maintaining the diagnosis accuracy, SpiderMon only targets switches relevant to the observed performance problem as detailed in §3.2.1. Second, SpiderMon collects the relevant telemetry data within a short time such that each switch only needs to keep a small amount of historical telemetry data as detailed in §3.2.2.

3.2.1 Relevant Switches Notification

#1: Only collect data after problem detection. Compared to other systems which collect data to a centralized collector all the time [6, 16, 32, 48], SpiderMon uses a default-off

collection strategy to minimize overhead. After the problem is detected, a special 'spider' packet is generated to notify relevant switches and start the telemetry collection on those switches. A "spider" packet carries: 1) an event_ID, which concatenates the switch ID and the event index to uniquely identify the problem, and 2) the 5-tuple of the victim flow. Spider packets are generated by mirroring the packet that triggered the diagnosis and recirculating it for transmission, while the original packet transmits as normal. To prevent possible packet drops during the transmission, all "spider" packets are prioritized in the network for lossless transfer.

#2: Only collect data from relevant switches. Instead of collecting telemetry from all switches, SpiderMon identifies the switches that are relevant to the detected problem by tracking packet-level provenance; it only retrieves data from these switches to minimize overhead. Packet-level provenance is modeled as $G := (V, E)$ for a detected event and the corresponding causality relations. G is a directed acyclic graph, where each node v represents an event, and each directed edge $e = (v_1 \rightarrow v_2)$ represents that v_1 leads to the event v_2. For latency problems in a network, all wait-for contentions in the switch queues are considered events in the provenance data. Since events at the upstream switches affect the events at the downstream switch, such upstream events are also incorporated into the provenance model. In this way, we can construct a provenance graph for a performance problem. By analyzing the locations of events, SpiderMon can select switches relevant to the specific problem.

#3: Track the provenance graph in the data plane. Unlike the central controller that Trumpet uses to inform relevant nodes, SpiderMon performs this procedure entirely in the data plane to reduce the latency of notifying relevant switches. It only requires switches to maintain telemetry data for a shorter time for the recent interval without losing necessary data. To achieve this, SpiderMon repeats the following two steps on each switch that receives the "spider" packet: 1) sends a traceback "spider" packet along the historical path of the victim flow, where the path is obtained using a bloom filter, 2) sends branch-search "spider" packets to ports that sent traffic and contended with the victim flow, where the ports are identified by a per-port traffic meter. Switches drop spider packets with duplicate IDs to avoid unnecessary processing (§C.1).

Timeout bloom filter. SpiderMon uses a timeout bloom filter (TBF) to track the victim flow's historical path. Regular bloom filters allow the insertion and the membership test of a flow ID. However, they can only support insertions, and the false positive rates increase with the number of inserted flows. A rotating bloom filter, on the other hand, can instantiate one instance per epoch, so that older data can be safely discarded; however, this is very coarse-grained as it only supports per-epoch deletion. To address those problems, SpiderMon adds a timeout feature to remove unneeded data from the bloom filter; this method provides a "sliding window" of historical flow information. For a switch with N ports, each egress

---

Algorithm 1: Timeout bloom filter data structure

Input: B: Timeout bloom filter, $inPort$: Incoming port index, $5-tuple$: 5-tuple, $curr\_TS$: Current timestamp, $epoch$: Timeout epoch

1 **Function** updateBF ($inPort$, $5-tuple$):
2      $hashValues = HASH (5-tuple)$
3   **for** $hashValue \in hashValues$ **do**
4          $B[hashValue][inPort] \leftarrow curr\_TS$
5   **return**

6 **Function** checkBF ($inPort$, $5-tuple$):
7      $hashValues \leftarrow HASH (5-tuple)$
8   **for** $hashValue$ in $hashValues$ **do**
9          $stamps \leftarrow B[hashValue][inPort]$
10   **if** $curr\_TS - stamp > epoch$ **then**
11   **return** False

12   **return** True

---

pipeline maintains a bloom filter group with M rows and N cells per row, and each column represents a bloom filter for the corresponding port. The TBF replaces the bit record with a short timestamp, which can be used to recognize the outdated records when querying the TBF. The details about maintaining and querying the TBF are shown in Algorithm 1, Figure 2(a) and Figure 2(b). The memory footprint of TBF can be reduced by shrinking the size of stored timestamps (§C.2).

Most recent, per-port traffic meter. SpiderMon identifies the relevant ports that contribute to high latency. To distinguish an ingress port with low throughput, SpiderMon maintains a traffic meter for each ingress port's traffic volume in the most recent time. Normal traffic meters in the switch are reset to 0 periodically, leading to information loss. Therefore, SpiderMon divides the time window into several small windows and associates those meters' values to realize a sliding window of the traffic amount within the most recent time window (details in §B).

#4: Reduce the collected telemetry data by pruning the provenance graph. Some causality relations are more important than others. SpiderMon leverages this to reduce overhead without sacrificing diagnosis accuracy. Specifically, if the traffic volume from some ingress ports is significantly lower than others, it is excluded from the possible root causes; so switches that contribute minimally to the problems are ignored. SpiderMon provides a tunable threshold and only sends spiders to the ports with high traffic rates. The robustness of this threshold is shown in §4.3.

To illustrate the relevant switch notification procedure, we use Figure 3 as an example of a multiple contention scenario. The high latency is detected at switch 0. Then the traceback "spider" is sent to the reverse path of the victim flow, namely, switches 1, 2, and 3. At the same time, the branch-search "spider" is sent to switches 4 and 6, with switch 5 being ignored due to the small traffic volume. If the traffic from switch 4 came from two other switches has sufficient volume, the branch-search "spider" packets will also be sent to those ports.

(a) Update timeout bloom filter   (b) Test timeout bloom filter   (c) Per-port Traffic Meter   (d) Telemetry data structure
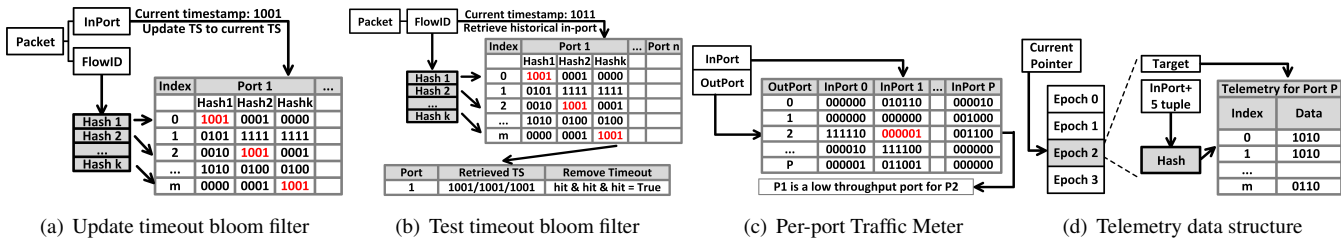
Figure 2: SpiderMon data structures
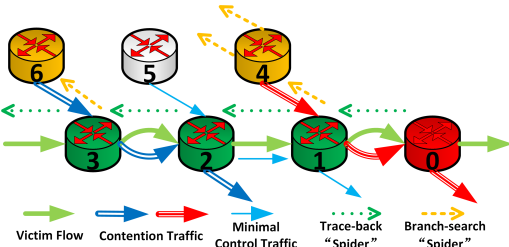
Figure 3: "Spider" packets propagation

3.2.2 Telemetry Data Collection

#1: Collect per-epoch per-flow information. Per-packet telemetry incurs a very high overhead and usually is unnecessarily fine-grained for diagnosis. SpiderMon records the history with a per-epoch flow-level log, which is stored in the switches' egress pipeline and each egress port has its own log. Dividing into epochs this way allows SpiderMon to observe changes among epochs. Each switch keeps a fixed number of epochs on the data plane and keeps the most recent ones in a circular buffer. When reporting the telemetry data, information of all epochs will be sent to the analyzer.

SpiderMon collects 36 bytes of data per flow, including the flow's 5-tuple, sequence number range, total traffic volume, total packet count, total queuing depth, the priority of the flow, and the incoming port. The network operators can add extra flow-level information in the telemetry data structure for diagnosing other network problems. The total amount of telemetry data varies with the flow arrival rate. To update, SpiderMon first identifies the right telemetry table based on the outgoing port, then hashes the flow ID to assign a slot in the telemetry data structure for that flow. By doing a bit-wise XOR between the packet's 5-tuple and the 5-tuple in the slot, we can determine whether this packet belongs to the existing flow by checking whether the result is 0. If so, this packet will be used to update this entry; otherwise, it will be considered as a new flow and replace the old one. The old entry will be packed and sent to the control plane for storage.

SpiderMon must maintain telemetry data for a minimum duration to ensure that the needed evidence for diagnosis is available, and this duration can be estimated as follows. Denote the threshold for detecting an unacceptable cumulative delay as T and the maximum round-trip propagation delay across the network as RTT. The time it takes to propagate spider packets from the initiator to relevant switches—recall that spider packets have high transmission priority and do not wait for normal traffic—is half RTT in the worst case. Since the problem is detected after accumulated delay exceeds T, the time duration a switch must maintain telemetry data to diagnose this problem is, therefore, $T + \frac{RTT}{2}$. The common RTT and T in the data center network is 0.5-2 ms and 10-15 ms respectively [15], so it would be more than enough for SpiderMon to preserve history for 20 ms.

#2: Provide synchronization among switches using flows' sequence number. The host-based solution cannot replay accurately, one of the reasons is the various network delay for packets, namely, the order of packets is not preserved at switches. SpiderMon has a similar problem when choosing the most relevant epoch on different switches for analysis. The correct epoch for the switch that triggered the problem is no doubt the most recent epoch, but for other switches on the historical path, the delay from the queuing and propagation may have caused the most relevant epoch to become a historical epoch. To solve this, SpiderMon keeps track of the [min_seq, max_seq] for each flow, and uses the victim flow's sequence numbers to find the correct epoch with the maximum overlap with this sequence number interval for the relevant switches.

#3: Trigger telemetry packet generation in the data plane. Unlike NetSight that uses mirroring for collection, SpiderMon uses the packet generator to report the per-epoch per-flow log to the root causes analyzer. The packet generator can be directly triggered in the data plane to minimize latency. Compared to retrieving the data via the switch control plane as in several previous works [27], SpiderMon is much more agile because it bypasses the low bandwidth and high latency connection between the data plane and the control plane.

The telemetry packet header contains 1) an event ID for identifying the performance problem; 2) a switch ID; 3) a partition index of the telemetry data; 4) a part of the telemetry data. The telemetry packets are generated by the packet generator on a programmable switch. The generated packets only have Ethernet and IPv4 headers without the payload for bandwidth savings. The IPv4 destination address of telemetry packets is set to the root cause analyzer so that the network will forward the packets to the analyzer. There is a maximum amount of telemetry data that can be inserted into a single packet, which is around 200 bytes due to the limitation of the PHV fields for the programmable switches. So the packet generator will generate a fixed number of telemetry packets according to the size of the telemetry tables.

Algorithm 2: Replay the queue condition

Input: T: the epoch period; N: flow packets count, s: time for the last packet
Output: $time\_list$: time list for the packets

1 **for** $t \in N$ **do**
2      $t \leftarrow s + \frac{T}{N}$
3      $time\_list \leftarrow time\_list + t$
4 **return** $time\_list$

#4: Only collect the telemetry data from relevant ports to reduce overhead. When a switch receives a spider packet from a certain port, usually only the telemetry data for that port will be reported to the analyzer, which reduces the amount of data collected.

3.3 Root Cause Analysis

SpiderMon develops a diagnosis strategy that is generalizable to diverse root causes with high precision and recall.

Efficiently localizing network problems and accurately identifying the root causes can be difficult, especially when the network conditions are dynamic and complex. Firstly, a good diagnosis algorithm needs to understand flow interactions and find the corresponding flows that occupied the queue. Secondly, once the problem has been localized, the diagnostic algorithm needs to further identify each problematic scenario with one or more root causes, such as micro-bursts or transient loops. However, most existing diagnostic algorithms do not have a clear boundary between those two steps. The identifications of the root causes are based on the matching of the problem patterns and observations, leading to slow diagnosis time and reduced diagnosis accuracy.

SpiderMon addresses these challenges with a two-step diagnostic algorithm: 1) efficiently analyze the queuing information at both flow level and aggregate level to recall all the problematic flows using wait-for graphs (WFG), as discussed in §3.3.1; 2) apply signature matching between the problematic flows and the root cause type, as described in §3.3.2.

3.3.1 Find the Possible Root Causes

To find all possible root causes with a high recall rate, SpiderMon uses WFG at both flow-level and aggregate-level to identify the abnormal behaviors from the telemetry data.

Wait-for relation. *If a packet from flow A enters a queue where the packets from flow B already exist in the queue, then flow A waits for flow B at this queue.*

Flow-level wait-for graph (WFG). *Each vertex represents a flow, and a directed edge from vertex A to vertex B represents that flow A waits for flow B.*

Wait-for weight. *Each directed edge's weight is calculated as follows: for a packet p_k from flow A, if x_k packets from flow B exist in the queue when p_k enters, then flow B blocks flow A with weight x_k. For all n packets from flow A during a certain period, the average weight $\frac{1}{n} \cdot \sum_{k \in [1,n]} x_k$ is the wait-for weight for the directed edge from vertex A to vertex B.*
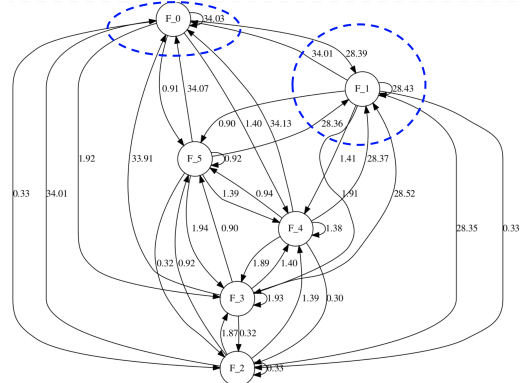
Figure 4: Identify the main contributors in WFG

Algorithm 3: Wait-for Graph Construction

Input: Seq: A sequence of packet, $level$: flow or port
Output: G: Wait-for graph for the given sequence

1 **for** $i \in [0, Seq.length]$ **do**
2   **if** $level=flow$ **then**
3          $Seq[i].vertex \leftarrow Seq[i].flow$
4   **else if** $level=port$ **then**
5          $Seq[i].vertex \leftarrow Seq[i].port$
6   **if** $Seq[i].vertex \notin G$ **then**
7          $G.AddVertex(Seq[i].vertex)$

8 **for** $i \in [0, Seq.length]$ **do**
9   **for** $j \in [0, pkt.qdepth]$ **do**
10          $edge \leftarrow (Seq[i].vertex \Rightarrow Seq[i-j].vertex)$
11          $G.AddEdgeWeight(edge, 1)$

12 **return** G

Aggregated wait-for graph. *SpiderMon also aggregates the flow according to the source IP, incoming port, or other keys to construct aggregated-level WFGs to find root causes other than flows' misbehavior. One typical example used in SpiderMon is the port-level WFG.*

After receiving all the telemetry data from the switches, SpiderMon uses the gap-based sampling strategy [25] to replay the queuing condition on the switch (Algorithm 2). The actual sequence of the packets is not important since we only need the generated wait-for graph to be similar.

To find the main contributors for the queuing, we rely on the wait-for graphs to show the provenance relations between contending flows. For each queue, SpiderMon will construct flow-level WFGs and port-level WFGs as in Algorithm 3, which will be used to determine the main contributors. Basically, to identify the main contributors of the queue is to divide the flows in the queue into victims (suffer from queuing) and main contributors (contribute to queuing) and maximize the wait-for relations between those two groups. SpiderMon is able to show that this division can be easily derived by the following Theorem 1, and identify the main contributors as in Algorithm 4. We prove Theorem 1 in Appendix §A.

Degree of the vertex. *Sum of all incoming edge weights subtracts the outgoing edge weights.*

Algorithm 4: FindContributor

Input: G: Wait-for graph for the given sequence
Output: $ctrs$: A set of main contributors

1 **for** $X \in G$ **do**
2      $D(X) = \sum_{e \in \{<i,j>|j=A\}}^{e} w_e - \sum_{e \in \{<i,j>|i=A\}}^{e} w_e$
3   **if** $D(X)>0$ **then**
4          $ctrs \leftarrow ctrs + X$

5 **return** $ctrs$

Theorem 1. *The wait-for relation between two groups, divided by one cut, is maximum, if and only if one group only contains positive degree vertices while the other contains only negative degree vertices.*

Figure 4 is an example scenario of micro-burst with flows 0 and 1 as the burst flows, and both of them have been identified by the algorithm as the main contributors.

3.3.2 Precisely Identify Root Causes

To precisely identify the reason behind the main contributors determined in the first step, SpiderMon relies on signature matching to recognize different root causes. We give four signatures for four common root causes in Algorithm 5, using both telemetry and network configuration information. The signatures can be extended if more root causes are added. For better illustration, we consider the scenarios in Figure 1 and show the signatures in Figure 5. A detailed signature definition can be found at §G.

Micro-bursts. SpiderMon can identify all the main flow-level contributors at different hops along the victim flow's historical path. As shown in Figure 5(a), the micro-burst flow has many wait-for edges with large weights pointing to itself due to a large amount of traffic during the problematic time.

Different priorities. For contention between flows with different priorities, SpiderMon checks the priority of the victim flow and the main flow-level contributors. The contributor flows with higher priority compared to the victim flow can be identified as the root causes, as shown in Figure 5(b).

ECMP load imbalance. For the load imbalance problem displayed in Figure 1(b), SpiderMon will find the flow-level main contributors and check if they are routed by ECMP. Then SpiderMon calculates the ECMP imbalance ratio with the throughput of all flows routed by ECMP rules, using the traffic volume provided by per-flow telemetry data. The problematic ECMP groups can be identified when the calculated ratio is largely imbalanced as in Figure 5(c).

Transient/persistent loops. For the latency problem caused by transient or persistent loops as shown in Figure 1(c), SpiderMon searches the port-level contributors along the contributor traffic's path. If the same port is observed twice during the search procedure, all those ports have a high possibility to form a loop for specific traffic. Furthermore, the flow ID will be checked to further confirm the transient/persistent loop.

Algorithm 5: Root Causes Diagnostic Algorithm

Input: f_WFG: flow-level WFG, p_WFG: port-level WFG, T: Telemetry information, K: Network topology and configuration

```
1  /* Diagnose flow-level problems              */
2  for sw ∈ Switches on victim's path do
3      f_CTR_sw ← FindContributor(f_WFG_sw)
4      for f ∈ f_CTR_sw do
5          // Is micro-burst?
6          check flow f throughput
7          // Is priority problem?
8          check flow f priority
9          // Is routed by ECMP rules?
10         check aggregated throughput for ECMP switches

11 /* Diagnose port-level problems              */
12 for sw ∈ Switches on victim flow's path do
13     p ← victim flow's outgoing port
14     CheckPort(p,{})

15 /* Recursive function for port-level         */
16 Function CheckPort(p, p_set):
17     // Does routing contain loop?
18     check whether there is a loop
19     // Search dominant port contributors
20     p_CTR_sw ← FindContributor(p_WFG_sw)
21     for p′ ∈ p_CTR_sw do
22         // Check the related port
23         src_p ← the port connect to port p′
24         CheckPort(src_p, p_set + p)
```

---

4 Evaluation

Next, we evaluate SpiderMon along several dimensions: diagnosis effectiveness, overheads, and robustness.

Setup. Our hardware testbed deploys SpiderMon to a Barefoot Tofino switch, written in 1147 lines of P4-Tofino code, to evaluate the switch-level performance. The switch is logically partitioned to emulate a topology with multiple logical switches; logical links are emulated by port-to-port connections using direct attach cables. The switch is also physically connected to eight servers through 25 Gbps links. The switch has $32\times$ 100Gbps ports, and each can be configured as four 25Gbps ports with a breakout cable; each server has two six-core 3.4GHz CPUs, 128GB RAM, and one 25Gbps NIC. In addition, we have set up a simulation environment that uses the BMv2 software switches in the NS3 simulator with 945 lines of P4 code running on CloudLab servers, evaluating the network-level performance. Each server has an eight-core 2.0GHz CPU and 32GB RAM. A K=4 standard fat-tree topology with 20 switches and 32 hosts is simulated with 1 Gbps link bandwidth. We also implement the root causes analyzer with 843 lines of Python code.

Workloads. We simulate empirical workloads from production networks for our evaluation. The flow size distribution is taken from three different traces: web search [5], cache [35], and Hadoop [35]. The arrival time of different flows is based on a Poisson process and the flow arrival rate is varied to obtain different load utilizations in the network. The source and destination for each flow are chosen uniformly at random.

All flows are TCP.

Baseline systems. We compare SpiderMon against five baseline solutions. 1) **Trumpet** [31]: a trigger-based reactive host system. When it detects a problem requiring network-wide information on one host, the controller will collect data from related servers upon a trigger. This incurs a latency of at least an RTT. 2) **NetSight** [17]: an in-network system that proactively collects 'postcards' for each packet from the switches. 3) **Marple** [32]: a query-based in-network system, which is deployed to all switches using monitoring queries that a) detect high latency, b) query packet counts, and c) perform 'EWMA over latencies'. 4) **Pathdump** [37] and **SwitchPointer** [38]: two proactive, network+host solutions. Pathdump tracks paths and performs diagnosis on end-hosts, and SwitchPointer further tracks packet epochs in the network.

4.1 Diagnosis Effectiveness

We evaluate the diagnostic effectiveness of SpiderMon using multiple scenarios.

1. Micro-bursts are created by injecting 5 short-lived (10-100 μs) UDP flows from SW0 to SW1 and from SW2 to SW3 as in Figure 1(a). The throughput of micro-burst flows is set to 90%×line-rate. **Diagnosis:** Fig. 5(a) shows the combined wait-for graph at two switch ports generated by SpiderMon, which shows that the two micro-burst flows E and H dominate the queues and are the only two main contributors with positive degrees. The other 3 UDP flows are not included in the WFG since they end before the victim flow starts or start later than the 2 contending UDP flows.

2. Priority contentions inject 5 high-priority TCP flows with priority queuing from SW0 to SW1 and from SW2 to SW3 as in Figure 1(a). **Diagnosis:** As Figure 5(b) shows, flow C and D are the main contributors to the congestion with higher priority and larger degrees. Other priority flows have no interference with the victim flow so the WFG excludes them.

3. ECMP imbalance scenarios randomly pick a switch (except core switches) and split traffic to two uplink ports with 4:1 imbalanced load. The ECMP group imbalanced lasted for hundreds of microseconds. **Diagnosis:** When we find the main contributors to the queuing, SpiderMon will check whether they are routed by ECMP policy. In Figure 5(c), both main contributors (flow C and D) are routed by ECMP rules on switch 0, so SpiderMon uses the telemetry information for switch 0 and computes the number of flows and traffic amount sent to each ECMP port. If the number of flows or traffic amount within that epoch is largely imbalanced, then there is an issue with the ECMP rules or hash functions.

4. Loops create a 4-hop routing loop with 2 aggregation switches and 2 core switches as in Figure 1(c). The routing loop only affects a small group of flows and the problem only lasts for 100 μs. **Diagnosis:** Port-level WFGs identify a loop as the root cause: the victim flow is reported on switch 8 port 1 so that the WFG leads us to the main contributor, port 0. Since SW8-P0 receives traffic from SW4-P0, we further construct a WFG for SW4-P0 and determine another main contributor. With this recursive searching procedure, SpiderMon finds that the port-level contributors form a loop and the traffic belongs to the same group of flows.

5. Complex problem diagnosis. Next, we test a diagnostic scenario with multiple problems. In Figure 6, the victim flow contends with a micro-burst flow at switch 1, a high priority flow at switch 7, and high-volume traffic caused by ECMP imbalance at switch 5. First, SpiderMon constructs the WFG with the collected information for the problem and identifies 5 flows (flow C, E, F, J, and L) with positive degrees. Next, SpiderMon checks the property of each such flow and identifies flow C as a micro-burst flow without any congestion control, while flow J is a flow with higher priority than any other flows crossing those switches. Then it checks the amount of the transmitted traffic in the same epoch and identifies flows E and F to be related to an ECMP imbalance. However, flow L is removed from the root causes; it is a normal TCP flow since its degree is small and there is no further evidence from the telemetry information to show that this flow is problematic.

6. Sporadic & transient problem diagnosis. We also evaluate multiple diagnostic situations with sporadic and transient problems. The traffic workloads are generated from random sources and destinations, and the problems could happen at different locations in the network randomly with short-lived root causes. Take the micro-burst experiment as an example. A high throughput UDP flow is introduced between a random source and destination at a random time, lasting for 100 μs. The experimental results shown in Section 4.2 are generated with sporadic problems for each scenario.

4.2 Comparison with Baseline Systems

Precision and recall. We first show the precision and recall rate for different solutions, by tuning the parameters of each system so that it can achieve the best performance for each scenario. Those include the maximum tolerable link load imbalance ratio, link utilization, per-flow throughput, and so on. Details about each scenario's parameters are in §F. Here we show the results for web trace only, the results for cache and Hadoop traces are included in §E.2. For the web trace, 30% of the flows are 1–30MB, so that multiple large flows can be concurrently active from/to one switch port.

As shown in Figure 7, Trumpet cannot achieve both high recall and accuracy at the same time for the transient congestion since it can only infer the in-network condition based on the calculated link utilization and end-to-end delay. Due to the different network delays and packet loss, the evidence for the transient problems may be inaccurate and unreliable on the host. Trumpet also fails to diagnose the ECMP imbalance problem because it does not have path information for every flow to identify the traffic split at the ECMP switches. Trumpet also fails to diagnose the loop problem because packets involved in loops do not reach the hosts, leaving no evidence for Trumpet to find out the root cause.

(a) Micro-burst Contention   (b) Priority Contention   (c) ECMP Load Imbalance   (d) Loop
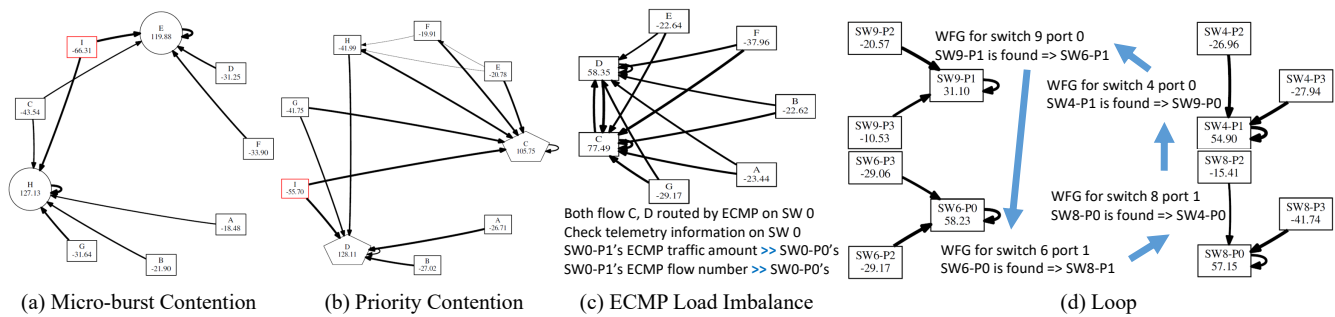
Figure 5: Example wait-for graphs of several root causes. Each box (TCP flow/port), circle (UDP flow), and pentagon (High priority flows) represent one flow or port, and the port name is described according to Figure 1(c). Bolder edges represent heavier wait-for relations, edges with small weights are tailored. The number under the flow/port name shows the node degree, and positive degrees will be identified as main contributors.
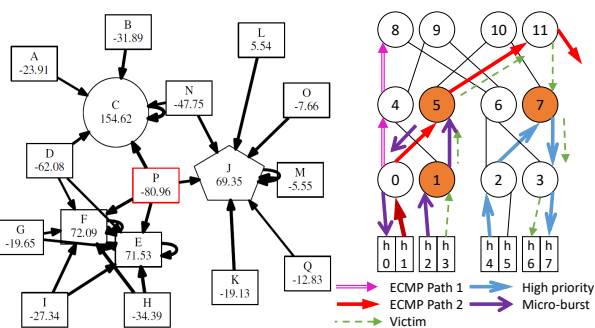
Figure 6: The WFG for victim flow P, with a micro-burst, a priority-related contention, and an ECMP imbalance at different hops.

Marple falls short in diagnosing transient contention like micro-bursts. This is because Marple enables queries only when needed, so it collects data reactively, which incurs an additional latency. The per-hop queuing information is only collected when the accumulated queuing latency exceeds the threshold. This control loop delay leads to information loss for transient problems—when the system begins collecting data from a switch near the destination, the transient bursty flow at a previous hop may have already ended. Only Marple and Trumpet are reactive systems in our evaluation.

PathDump and SwitchPointer both achieve relatively good performance. PathDump carries path information along with the packets, and SwitchPointer upgrades PathDump with switch data that records the flows that travel the same switch in the same epoch, which outperforms PathDump. However, both of them failed to identify transient problems since they lack queuing information—they instead recompute link utilization using packets received at end hosts. If a large amount of packets are dropped in the network due to congestion loss or TTL expiration, it would be very hard to reconstruct the transient network condition. Another interesting fact is that both solutions add extra in-network mechanisms (path tracking [37]) to detect the routing loop, so they both achieve great performance in detecting and diagnosing loops.

NetSight achieves the second-best performance since it

(a) Micro-burst   (b) Priority contention

(c) ECMP imbalance   (d) Loops (solutions except Marple and Trumpet overlap at the top right corner)
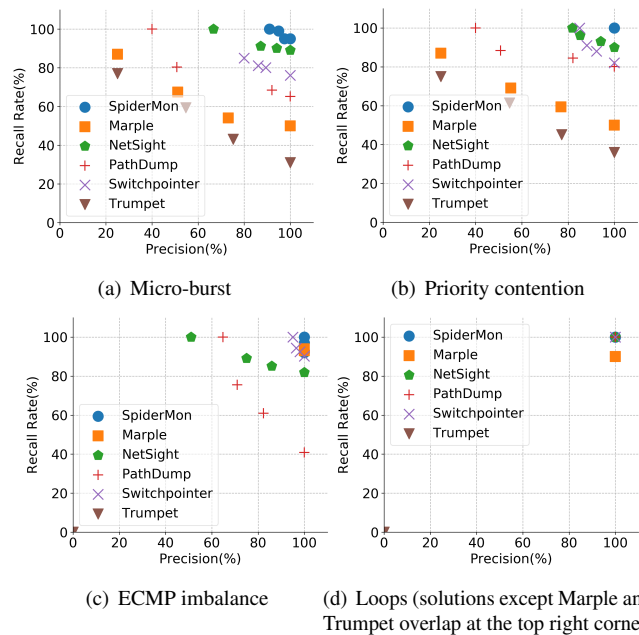
Figure 7: Diagnostic effectiveness for different solutions

collects per-packet postcards. One drawback is that to keep overhead down, NetSight omits important data like packet priority or precise timestamps. Instead, it uses topology information to place the postcards in order. However, information that describes how flows interact cannot be obtained, which is essential for diagnosing transient problems.

SpiderMon is able to achieve nearly 100% recall and precision for all tested scenarios. The reason is that SpiderMon collects accurate packet-level information within a time interval. For micro-burst and priority flow contention, each flow's throughput within the same epoch where congestion happens will be recorded and reported in the telemetry data; for the ECMP imbalance problem, the flow ID and output port will be recorded, so that the ECMP imbalance ratio can be calculated; for the loop problem, the loop can be easily detected in the procedure of WFG construction.

To summarize, host-based solutions (Trumpet, PathDump
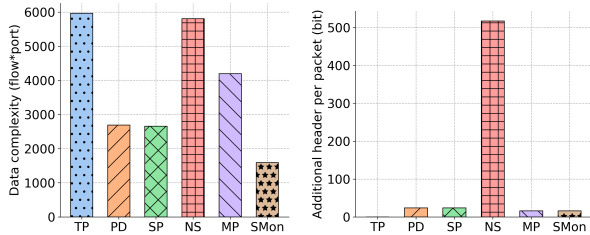
(a) Diagnostic data complexity   (b) Additional bandwidth overhead

Figure 8: Diagnostic data complexity for different systems; the additional per-packet header shows the bandwidth overheads for Trumpet (TP), PathDump (PD), SwitchPointer (SP), NetSight (NS), Marple (MP), and SpiderMon (SMon).

and SwitchPointer) all lack accurate in-network information, like accurate queuing information and the packet loss for traffic other than TCP (they can only observe packet loss at the sender with the help of TCP's congestion control). As for the proactive in-network approach in NetSight, it scarifies the telemetry data granularity to keep overhead low. Only the packet header, switch ID, output port, and a version number are included. It uses topology information to assemble out-of-order postcards since the fine-grained timestamps and queuing information are not included in the postcards. The reactive in-network Marple system can potentially collect the information at very fine granularity but it can only start this reactive network-wide query after a half-RTT delay after the problem has been detected. The experiments over Cache and Hadoop traces have qualitatively similar results with the web search trace; more details can be found in §E.2.

Diagnostic overhead. To evaluate the diagnosis complexity and resource usage of different solutions, we measure the amount of collected data and the extra bandwidth requirements. We measure the diagnosis complexity using the amount of telemetry data stored and used in the diagnostic procedure, using (flow×port) as the unit to denote the complexity of flow information collected at switch ports. Since the host-based solutions collect information from the end hosts, and they reconstruct the utilization of different links [37], we multiply the average path length with the flow×host as the overall complexity. Both switches and hosts have limited storage spaces and may restrict the scalability of the solutions. Under the same scenario for diagnosing micro-bursts, we show the amount of telemetry data for different systems in Figure 8(a). Reducing the diagnosis complexity not only relieves the burden to process the collected information for the central controller but also saves the storage space to store the diagnostic data for future usages.

Trumpet processes packets and match triggers in real time during the monitoring phase, so no packet is stored. But in the reactive data gather-report phase, data from multiple hosts will be reported. In order to construct every link utilization, the throughput of all flows will be reported and stored for
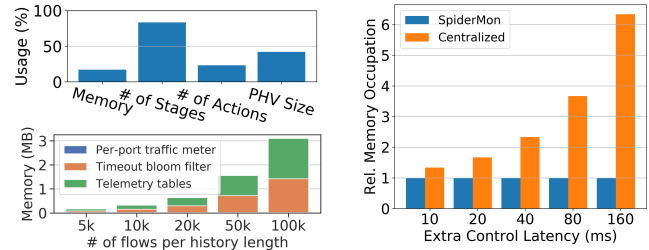
(a) The resource usage on Tofino switch is low. Per-port traffic meter is too small to be visible in the figure.

(b) Relative memory usage under different controller latency with SpiderMon as the baseline.

Figure 9: Switch memory occupation

further analysis. Pathdump and SwitchPointer need to store per-packet history, since the problem may be detected after analysis. But both systems rely on the path information to find out the flows that travel the same link with the victim flow so that the data complexity can be reduced by filtering out irrelevant flows. Marple stores the query results from every switch to reproduce the scenarios, so such data will be transmitted as well as stored on the hosts. But Marple starts the collection after problem detection and stops after the problem disappears, collecting less but potentially incomplete data. NetSight stores all packet postcards and processes them in real time. All flows from all the switch ports are collected and stored, leading to a similar data complexity as Trumpet. SpiderMon only collects data after a problem is detected and only from relevant switches. Thus, the overhead for collecting telemetry data is much lower than the other systems.

Monitoring bandwidth overhead. Next, we measure the amount of extra bandwidth usage during monitoring. Trumpet never collects in-network data; it only uses the network to communicate with other servers, so it has a low overhead. PathDump and SwitchPointer both use two VLAN tags of 24 bits for path and switch epoch information. NetSight always collects per-packet postcards to the host for analysis, and the per-packet additional bandwidth occupation is 15 bytes/packet × average hop count because NetSight will generate a postcard for the packet at every hop. Marple introduces a 16-bit header to carry the per-packet end-to-end latency, and during the monitoring phase, it will group the packets with their per-hop queuing latency and sent them to the controller. SpiderMon adds a 16-bit monitor header to every packet when it enters the network, and removes it before forwarding the packet to the end-host as mentioned in §3.1.

Switch resource overhead. Figure 9(a) shows the switch resource usage of SpiderMon, which fits comfortably in a Tofino pipeline. It also shows how SpiderMon scales with the number of flows seen during a collection period. Modern data centers have millions of concurrent flows per switch, but since SpiderMon only keeps tens of milliseconds of history, the number of flows per epoch is much smaller. Switch memory size increases steadily over time [29], so SpiderMon can scale to even more flows with more recent hardware.
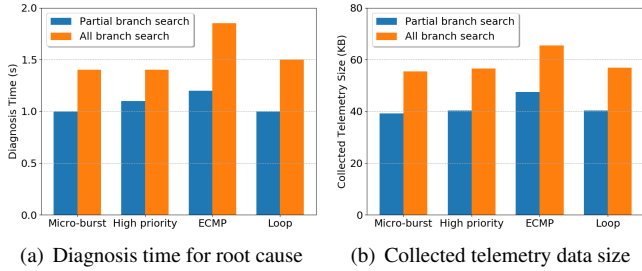
(a) Diagnosis time for root cause (b) Collected telemetry data size
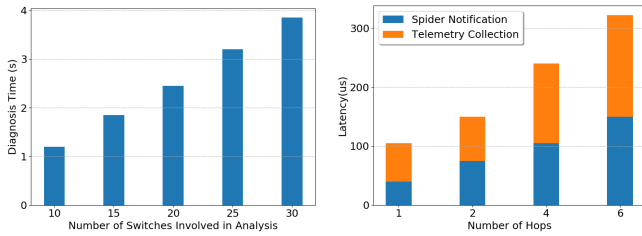
(c) Diagnosis time with different num- (d) The latencies for "spider" packets
ber of switches   and telemetry

Figure 10: Branch-search metrics for SpiderMon

To show the benefit of informing related switches in the data plane in a distributed manner, we compare SpiderMon with a centralized reactive strawman system, which uses a centralized node to receive the detected problems, identifies the related switches, and retrieves data from them. We vary the additional latency that this centralized controller introduces. Figure 9(b) shows that this solution requires more memory to store a larger amount of historical data to avoid the loss of relevant evidence for diagnosis. In comparison, SpiderMon only needs to preserve the history within the maximum queuing latency + half RTT (§3.2.2).

4.3 Diagnostic Robustness

We finally evaluate the diagnostic robustness of SpiderMon using different metrics related to branch-search coverage, epoch length, and cumulative latency. Within a range of adjustments, SpiderMon can diagnose the performance problems with ideal precision and recall. Network operators are allowed to adjust the parameters of SpiderMon according to their requirements.

Overall methodology. SpiderMon empirically adjusts the parameters under different network loads. Given a particular network traffic load, operators could systematically test the precision and recall rates of SpiderMon with different metric choices. Suitable choices should strike a good balance between the recall rate and the size of collected telemetry data for throughput metrics, switch memory consumption for epoch metrics, and the sensitivity of problem detection for latency metrics. The optimal parameters vary under different network loads. We provide the results of parameter adjustments using our experimental settings in the following, while network operators could follow the same methodology to obtain their preferred parameters.

Branch-search threshold. SpiderMon provides different options for spider packet propagation in terms of its reach (e.g.,
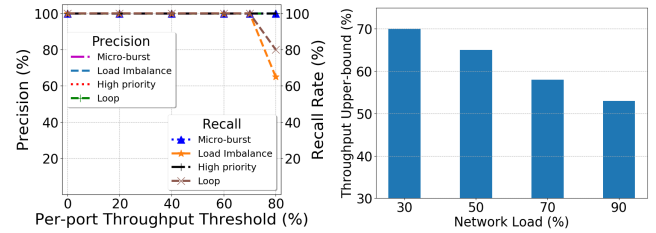
(a) Precision & recall rate for the root (b) Upper-bound of throughput thresh-
causes with 30% load   old

Figure 11: Throughput metrics for SpiderMon

all or some branches). Figure 10(a) and Figure 10(b) provide comparisons with different options on both the diagnosis time of root cause analysis and the size of collected telemetry data. Note that the number of relevant switches in SpiderMon is generally much smaller than the total network size since SpiderMon uses the wait-for relation and provenance model to precisely target only those relevant switches that contribute to the observed performance problem. Therefore, even with all-branches spider packets propagation (search all ports with > 0 throughputs), SpiderMon is efficient compared to more rudimentary diagnosis strategies that must comb through all data from all switches. Even for relatively widespread performance problems involving up to 30 relevant switches, it takes under 4 seconds to run the root cause diagnosis algorithm (Algorithm 5) on a 4.3GHz CPU, as shown in Figure 10(c). In addition, we evaluate the latency for spider packets propagation and the subsequent retrieval of the telemetry data, using 50 Gbps link bandwidth and 20μs link delay. From the results shown in Figure 10(d), we can see that a few microseconds are enough to perform the entire retrieval operation with arbitrary fat-tree topologies, no matter the choices of branch-search options. This is because SpiderMon's mechanisms run in the data plane. As a result, network operators can send "spider" packets without setting the branch-search threshold if the overhead can be tolerated based on their requirements.

We further evaluate the precision and recall rates under different branch-search coverage with different network loads. Figure 11(a) shows the results under 30% network load, indicating that the precision can always achieve 100% while the recall rates decrease if the threshold is too high. To trade-off the branch-search overhead and the recall rates, we suggest using 70% as the threshold in this case since it strikes a good balance. Following the same strategy, we summarize the upper bound of branch-search thresholds for operators to adjust under different network loads, as shown in Figure 11(b).

Epoch length. SpiderMon can change the length of the telemetry epoch to save memory but trade-off telemetry granularity. Network operators can adjust the telemetry epoch according to their requirements. Under different network loads, we provide the upper bound of the epoch length. For example, Figure 12(a) shows the results with the network load at 30%. We evaluate the precision and recall rates under different epoch lengths. The precision is always 100%, while

(a) Precision & recall rate for the root causes with 30% load

(b) Upper-bound of epoch length

Figure 12: Epoch metrics for SpiderMon

(a) Cumulative latency under different network loads

(b) Upper-bound of cumulative latency threshold

Figure 13: Latency metrics for SpiderMon

the recall rate decreases in some scenarios when the length of epoch exceeds 30 ms. We further measure the precision and recall rates under different network loads, and identify the upper-bounds of epoch length, as shown in Figure 12(b). The upper-bound epoch length used for telemetry collection decreases with increasing network load.
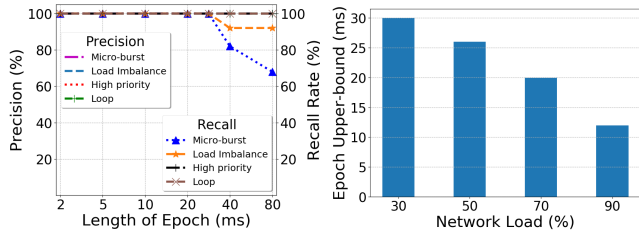
Cumulative latency threshold. SpiderMon provides a tunable cumulative latency threshold for problem detection, allowing network operators to customize problem trigger frequency for different applications. Figure 13(a) shows the CDF of different cumulative latency under different network loads in the absence of problems, where the cumulative latency is normalized by the maximum queuing latency of a single switch. Under different loads, the choice of cumulative latency threshold varies according to the trade-off between overhead and recall rate. The higher the sensitivity of the network to problem detection, the more switches are visited, and thus higher overhead. We further evaluate the recall rates of SpiderMon under different loads and summarize the upper bound of cumulative latency thresholds for reaching 100% recall in all scenarios in Figure 13(b).

5   Related Work

Switch-based telemetry. Telemetry systems such as Sonata [16], Marple [32], FlowRadar [26], *Flow [36], NetSeer [47] and Dapper [14] leverage programmable switches for fine-grained data collection. However, query-driven systems [16, 32] cannot dynamically change the targeted events at small timescales, and blanket monitoring systems [17,36] incur high collection overhead. SpiderMon aims to achieve lightweight yet accurate telemetry information collection. Two recent works, NetSeer [47] and PINT [8], share our high-level goal of reducing telemetry overhead. NetSeer detects per-flow performance events for compression, and PINT aggregates telemetry information across hops or flows to save bandwidth. Compared to these works, SpiderMon co-designs monitoring and posterior diagnosis based on wait-for relations for closed-loop diagnosis.

Diagnosis systems. SwitchPointer [38] and PathDump [37] collect both in-network and host data for diagnosis. Trumpet [31] monitors every packet at hosts and reports triggered events. SNAP [43] diagnoses network problems using logs (e.g., TCP statistics, socket calls) collected at hosts. How-

ever, these systems rely on a central controller and perform software-based monitoring. NetMedic [23], 007 [6], Net-Poirot [7] use statistical methods and/or machine learning to identify root causes. Network provenance [42] tracks how packets flow through a network and apply formal reasoning to identify root causes. Deter [25] can process and replay a TCP trace to diagnose performance degradation. Compared to these works, SpiderMon leverages the telemetry information from programmable switches, and it uses wait-for relations to reason about performance contention in-network. Our recent workshop paper sketches a similar roadmap [41], but it does not contain a concrete design, implementation, or evaluation.
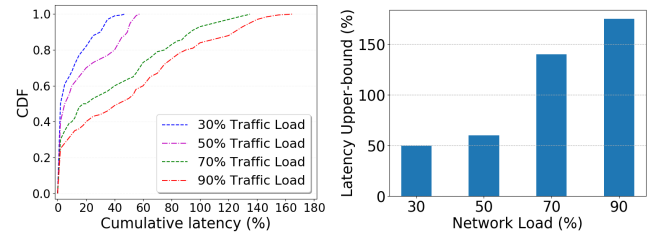
Monitoring. Another line of recent work focuses on designing compact data structures [11, 18, 19, 27, 44] with tradeoffs between accuracy and resource footprints. OmniMon [19] divides flow-level monitoring across different network entities to satisfy resource constraints. BeauCoup [11] supports multiple distinct counting queries simultaneously while requiring a small number of memory accesses. These data structures complement SpiderMon by reducing switch resource usage.

6   Conclusion

SpiderMon is a system that achieves high coverage and low overhead in monitoring and diagnosing network performance problems. It monitors every flow in the data plane and triggers diagnostic events upon problem detection. It precisely collects diagnostic information in an as-needed fashion. We prototype SpiderMon on Tofino hardware and BMv2 software switches and show that it can leverage wait-for relations to accurately pinpoint root causes for complex problems. SpiderMon also has low overheads for telemetry collection, switch resources, and network bandwidths.

Acknowledgment

References

[1] Network Congestion Management: Considerations and Techniques. https://www.sandvine.com/hubfs/downloads/archive/whitepaper-network-congestion-management.pdf.

[2] sFlow. http://www.sflow.org/.

[3] Solving the mystery of link imbalance: A metastable failure state at scale. https://engineering.fb.com/production-engineering/solving-the-mystery-of-link-imbalance-a-metastable-failure-state-at-scale/.

[4] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, A. Vahdat, et al. Hedera: Dynamic flow scheduling for data center networks. In *USENIX NSDI*, 2010.

[5] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *ACM SIGCOMM*, 2010.

[6] B. Arzani, S. Ciraci, L. Chamon, Y. Zhu, H. H. Liu, J. Padhye, B. T. Loo, and G. Outhred. 007: Democratically finding the cause of packet drops. In *USENIX NSDI*, 2018.

[7] B. Arzani, S. Ciraci, B. T. Loo, A. Schuster, and G. Outhred. Taking the blame game out of data centers operations with NetPoirot. In *ACM SIGCOMM*, 2016.

[8] R. B. Basat, S. Ramanathan, Y. Li, G. Antichi, M. Yu, and M. Mitzenmacher. PINT: Probabilistic in-band network telemetry. In *ACM SIGCOMM*, 2020.

[9] H. Chen, N. Foster, J. Silverman, M. Whittaker, B. Zhang, and R. Zhang. Felix: Implementing traffic measurement on end hosts using program analysis. In *ACM SOSR*, 2016.

[10] X. Chen, S. L. Feibish, Y. Koral, J. Rexford, and O. Rottenstreich. Catching the microburst culprits with Snappy. In *SelfDN*, 2018.

[11] X. Chen, S. Landau-Feibish, M. Braverman, and J. Rexford. BeauCoup: Answering many network traffic queries, one memory update at a time. In *ACM SIGCOMM*, 2020.

[12] J. Cho, H. Chang, S. Mukherjee, T. Lakshman, and J. Van der Merwe. Typhoon: An SDN enhanced real-time big data streaming framework. In *ACM CoNEXT*, 2017.

[13] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, et al. Azure accelerated networking: SmartNICs in the public cloud. In *USENIX NSDI*, 2018.

[14] M. Ghasemi, T. Benson, and J. Rexford. Dapper: Data plane performance diagnosis of TCP. In *ACM SOSR*, 2017.

[15] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z.-W. Lin, and V. Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *ACM SIGCOMM*, 2015.

[16] A. Gupta, R. Birkner, M. Canini, N. Feamster, C. Mac-Stoker, and W. Willinger. Network monitoring as a streaming analytics problem. In *ACM HotNets*, 2016.

[17] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *USENIX NSDI*, 2014.

[18] Q. Huang, X. Jin, P. P. C. Lee, R. Li, L. Tang, Y.-C. Chen, and G. Zhang. SketchVisor: Robust network measurement for software packet processing. In *ACM SIGCOMM*, 2017.

[19] Q. Huang, H. Sun, P. P. C. Lee, W. Bai, F. Zhu, and Y. Bao. OmniMon: Re-architecting network telemetry with resource efficiency and full accuracy. In *ACM SIGCOMM*, 2020.

[20] S. Ibanez, G. Brebner, N. McKeown, and N. Zilberman. The P4-> NetFPGA workflow for line-rate packet processing. In *ACM FPGA*, 2019.

[21] N. Jiang, D. U. Becker, G. Michelogiannakis, and W. J. Dally. Network congestion avoidance through speculative reservation. In *IEEE HPCA*, 2012.

[22] R. Joshi, T. Qu, M. C. Chan, B. Leong, and B. T. Loo. BurstRadar: Practical real-time microburst monitoring for datacenter networks. In *ACM APSys*, 2018.

[23] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and V. Bahl. Detailed diagnosis in enterprise networks. In *ACM SIGCOMM*, 2010.

[24] A. Khandelwal, R. Agarwal, and I. Stoica. Confluo: Distributed monitoring and diagnosis stack for high-speed networks. In *USENIX NSDI*, 2019.

[25] Y. Li, R. Miao, M. Alizadeh, and M. Yu. Deter: Deterministic TCP replay for performance diagnosis. In *USENIX NSDI*, 2019.

[26] Y. Li, R. Miao, C. Kim, and M. Yu. FlowRadar: A better NetFlow for data centers. In *USENIX NSDI*, 2016.

[27] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman. One sketch to rule them all: Rethinking network flow monitoring with UnivMon. In *ACM SIGCOMM*, 2016.

[28] A. Ludwig, J. Marcinkowski, and S. Schmid. Scheduling loop-free network updates: It's good to relax! In *ACM PODC*, 2015.

[29] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching ASICs. In *ACM SIGCOMM*, 2017.

[30] R. Mittal, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, D. Zats, et al. TIMELY: RTT-based congestion control for the datacenter. In *ACM SIGCOMM*, 2015.

[31] M. Moshref, M. Yu, R. Govindan, and A. Vahdat. Trumpet: Timely and precise triggers in data centers. In *ACM SIGCOMM*, 2016.

[32] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim. Language-directed hardware design for network performance monitoring. In *ACM SIGCOMM*, 2017.

[33] Y. Ran, X. Wu, P. Li, C. Xu, Y. Luo, and L.-M. Wang. EQuery: Enable event-driven declarative queries in programmable network measurement. In *IEEE NOMS*, 2018.

[34] J. Rasley, B. Stephens, C. Dixon, E. Rozner, W. Felter, K. Agarwal, J. Carter, and R. Fonseca. Planck: Millisecond-scale monitoring and control for commodity networks. In *ACM SIGCOMM*, 2014.

[35] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *ACM SIGCOMM*, 2015.

[36] J. Sonchack, O. Michel, A. J. Aviv, E. Keller, and J. M. Smith. Scaling hardware accelerated network monitoring to concurrent and dynamic queries with *flow. In *USENIX ATC*, 2018.

[37] P. Tammana, R. Agarwal, and M. Lee. Simplifying datacenter network debugging with PathDump. In *USENIX OSDI*, 2016.

[38] P. Tammana, R. Agarwal, and M. Lee. Distributed network monitoring and debugging with SwitchPointer. In *USENIX NSDI*, 2018.

[39] Y. Tang, Y. Wu, G. Cheng, and Z. Xu. Intelligence enabled SDN fault localization via programmable in-band network telemetry. In *IEEE HPSR*, 2019.

[40] H. Wang, R. Soulé, H. T. Dang, K. S. Lee, V. Shrivastav, N. Foster, and H. Weatherspoon. P4FPGA: A rapid prototyping framework for P4. In *ACM SOSR*, 2017.

[41] W. Wang, P. Tammana, A. Chen, and T. S. E. Ng. Grasp the root causes in the data plane: Diagnosing latency problems with SpiderMon. In *ACM SOSR*, 2020.

[42] Y. Wu, A. Chen, and L. T. X. Phan. Zeno: Diagnosing performance problems with temporal provenance. In *USENIX NSDI*, 2019.

[43] M. Yu, A. Greenberg, D. Maltz, J. Rexford, L. Yuan, S. Kandula, and C. Kim. Profiling network performance for multi-tier data center applications. In *USENIX NSDI*, 2011.

[44] M. Yu, L. Jose, and R. Miao. Software defined traffic measurement with OpenSketch. In *USENIX NSDI*, 2013.

[45] Q. Zhang, V. Liu, H. Zeng, and A. Krishnamurthy. High-resolution measurement of data center microbursts. In *ACM IMC*, 2017.

[46] F. Zhou, Y. Gan, S. Ma, and Y. Wang. wPerf: Generic off-CPU analysis to identify critical waiting events. In *USENIX OSDI*, 2018.

[47] Y. Zhou, C. Sun, H. H. Liu, R. Miao, S. Bai, B. Li, Z. Zheng, L. Zhu, Z. Shen, Y. Xi, P. Zhang, D. Cai, M. Zhang, and M. Xu. Flow event telemetry on programmable data plane. In *ACM SIGCOMM*, 2020.

[48] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and H. Zheng. Packet-level telemetry in large datacenter networks. In *ACM SIGCOMM*, 2015.

A Proof for Contributors Identification Algorithm

Definition 6. *Degree of vertex. In a WFG, the degree of vertex A is the sum of all the adjacent edges' weights w_e:*

$$D(A) = \sum_{\{e=<i,j>|i=A\|j=A\}}^{e} \alpha_e \cdot w_e \qquad (1)$$

where α_e is 1 when A is the sink of edge e and -1 when vertex A is the source.

Lemma 1. *For a WFG, the sum of all the vertex's degree is 0:*

$$\sum_{X \in V}^{X} D(X) = 0 \qquad (2)$$

Proof: the WFG is a directed graph where every edge is pointing from a vertex to another vertex in the graph, so each edge will add weight w to the sink vertex and weight $-w$ to the source vertex.

Definition 7. *Flux of cut. For a cut in a WFG, the vertex will be divided into two sets, S1 and S2. Given all edges in the WFG has a positive weight according to the definition, we denote the flux of this cut as:*

$$Flux(cut) = \left| \sum_{i \in S1, j \notin S1}^{e=<i,j>} w_e + \sum_{i \notin S1, j \in S1}^{e=<i,j>} -w_e \right| \qquad (3)$$

where e represents the edge from vertex i to vertex j

Though the sum of all vertex's degree is 0, we can always find a cut whose flux is maximum, representing the provenance relation between vertexes from those two groups is the strongest. The set with a positive degree considers as the main contributor to the queue, while the other set contains victims of the queue, like normal flows or small flows. To find this cut efficiently, we have shown the hints by the following lemmas and theorems.

Lemma 2. *The flux of one cut is just the absolute value of the sum of all vertices' degrees in either set.*

Proof: The absolute value of the sum of all vertices' degrees in one set (ASD) can be written as:

$$\begin{aligned}
\text{ASD} &= \left| \sum_{i \in S1 | j \in S1}^{e=<i,j>} \alpha_e \cdot w_e \right| \\
&= \left| \sum_{i \in S1 \& j \in S1}^{e=<i,j>} \alpha_e w_e + \sum_{i \in S1 \& j \notin S1}^{e=<i,j>} \alpha_e w_e + \sum_{i \notin S1 \& j \in S1}^{e=<i,j>} \alpha_e w_e \right| \\
&= \left| 0 + \sum_{i \in S1 \& j \notin S1}^{e=<i,j>} -w_e + \sum_{i \notin S1 \& j \in S1}^{e=<i,j>} w_e \right| = Flux(cut)
\end{aligned}$$

$$(4)$$

Theorem 1*. *The WFG cut with maximum flux will divide the vertices with positive degrees into one set and negative degrees into the other set.*

Given the sum of all vertices' degrees are 0, for any cut: $\sum_{X \in S1} D(X) = -\sum_{Y \in S2} D(Y)$, namely, the absolute sum of degree for two sets are the same. Thus, for the cut that divide all vertices with positive degrees into one set, by contradiction, we can easily prove this is the cut with maximum flux.

The flux represents the wait-for relation between two groups from a cut of the wait-for graph, and the degree represents the value of incoming edges weights subtracting outgoing edges weights so that Theorem 1 is proofed.

B Fine-grained Sliding Window

During the telemetry collection process, SpiderMon maintains bloom filter and per-port per-epoch data structures to trace back all the relevant switches. However, part of these structures (e.g. traffic meter) needs to be reset to 0 at the beginning of an epoch due to the limited resources of the switch data plane. Therefore, there will be some information loss at the beginning of an epoch, leading to the diagnosis algorithm being inaccurate. SpiderMon employs a fine-grained sliding window on the data plane to achieve high accuracy for the used data structures.

The sliding window strategy slices each epoch into multiple pieces, and it proceeds in two actions: an update action and a decrease action. To explain simply, we take the traffic meter in the per-port data structure as an example. Assume one epoch T is divided into n small time slots. There will be n sub-traffic meters and each of them aims at a single time slot. When a switch receives a new packet during the update phase, the switch will update the corresponding sub-traffic meter based on the current time slot, as well as the total traffic meter. For decrease action, when the oldest sub-traffic meter no longer exists in the sliding window, the value of the corresponding sub-traffic will be subtracted from the total traffic meter and that sub-traffic meter will be reset to 0. Network operators are able to tune the fine-grained sliding window according to their demands. Basically, the more time slots an epoch is divided into, the higher the accuracy that the system can achieve. On the other hand, the overhead of telemetry data structures can be reduced with fewer time slots.

C Resource Usage Optimization

C.1 Avoid Duplicate Detection

In the scenario of the performance problem, there are lots of packets from the victim flow suffering from high latency problems, but not all of them will generate a diagnostic event independently. SpiderMon sets a limitation on the interval between two diagnostic events generated by the same flow, meaning that during one congestion, only the first packet suffering from high accumulated latency will trigger the diagnostic event. To avoid receiving multiple audit requests for the same diagnosis event, the switches will drop the duplicate "spider" packets with the same event ID as well.

C.2 Data Field Compression

For the applications like SpiderMon built on top of the programmable switches, keeping track of some data fields in the packet header or on the switch memory is always required. Compressing those data fields in order to reduce the extra header size or switch memory occupation is critical to the application performance. SpiderMon provides a method to compress the size of the data by extracting the most significant bits. This idea can be widely applied to many recorded data in such systems, and here are two typical examples that use this strategy:

The timeout bloomfilter in SpiderMon requires storing a large number of timestamps for each slot in the bloom filter, which is very resource consuming and inefficient. The timestamp is usually stored with 48 bits on the switch and SpiderMon uses the timestamp to perform the timeout operation. Given that the only operation on the timestamp is the subtraction of two timestamps and compare the difference with the timeout period, we can easily observe that the only significant bits in the timestamp are the bits around the period. Take the timeout period as 1 ms as an example, the most significant bits in the timestamp are the 10th, 11th, and 12th bits from the right, representing 0.512 ms, 1.024 ms, and 2.048 ms respectively. By extracting these three bits from the original timestamps and comparing the difference with bit array 010, we can get an approximation of the exact value that is calculated with the original timestamp. Adding more bits on the left (*e.g.* 13th and 14th) can prevent us from the danger of overflow while adding more bits on the right (*e.g.* 9th and 8th) can help us obtain a more precise result of the subtraction. With this method, SpiderMon only needs to store 6 bits for each timestamp and reduce the memory usage of the timeout bloomfilter by 87.5%.

Another example is the queuing information carried by the packets in SpiderMon, which is used to detect the performance problem by comparing the accumulated delay with the maximum delay threshold. For a certain application, the maximum delay threshold may be 1 ms. Then when we calculate the accumulated delay, the most significant bits are 8th, 9th, and 10th bits from the right, representing 0.128 ms, 0.256 ms, and 0.512 ms respectively. If any bit on the left of the 10th bit is not 0, SpiderMon will trigger the problem immediately, since it exceeds the threshold with this single-hop delay. In this way, SpiderMon only needs to add an extra header with 4 bits to carry each delay field instead of 19 bits, shrinking the overhead from the extra header by 78.95%. Note that in evaluation, we use 8 bits for each data field to provide better accuracy.

D Implementation

We have implemented SpiderMon on a Barefoot Tofino switch with 1147 lines of P4-Tofino code and also a BMv2 version for NS3 and MiniNet environments with 945 lines of P4 code. We also implement the root causes analyzer on the end-host with 843 lines of Python code.

Figure 14 depicts different components in a switch and the workflow for different packet types. The event record is used for checking duplicate "spider" packets, and the telemetry counter for guiding telemetry packet generation. Those two data structures are placed in the ingress because they need to make decisions on whether to mirror packets in the traffic management unit. The per-port meter and timeout bloom filter provide provenance data to guide the propagation of the "spider" packets, and the telemetry data structure stores historical flow information for diagnosis. Those two data structures, along with the problem detection component, are placed in the egress pipeline because they may require queuing information, which is only available in the egress pipeline. Note that the per-port telemetry information is stored separately on the switch, but not necessarily one table per stage. One stage in SpiderMon can store multiple egress ports' telemetry information.

To implement SpiderMon, the egress pipeline is required to detect the problems, store telemetry information, and provide temporary provenance hints for "spider" packet propagation. For switch architectures like SimpleSumeSwitch [20] (NetFPGA), P4FPGA [40], and SmartNICs, SpiderMon can also be implemented by taking the next switch's pipeline as the "egress pipeline" of former switches to detect congestion and collect telemetry information. This design requires more communication among switches, so both the latency for diagnosing the problem and the link bandwidth used by SpiderMon would also increase.

As for the hardware switch resource, modern switches have increasing memory sizes [29], and more ports usually represent more on-chip memory, which, we shall demonstrate in §4, is more than sufficient to support SpiderMon.

E Additional Experiment Results

E.1 Header Bandwidth Usage

Packet Size (B)	1480	1000	500	100
SpiderMon (Gbps)	23.51	23.5	22.84	20.51
Baseline (Gbps)	23.65	23.5	22.84	21.87

Table 1: SpiderMon's maximum throughput is quite close to the baseline switch with only forwarding rule.

As the monitor header added by SpiderMon is removed before forwarding the packet to the end-host, the corresponding overhead of the additional header is very trivial. We use iPerf to show the maximum throughput of traffic with different average packet sizes on the Tofino switch equipped with SpiderMon in Table 1 and compare it with a baseline switch program with only basic forwarding rules. As expected, SpiderMon 's end-to-end throughput is nearly identical to the baseline, meaning that the bandwidth overhead of the monitoring phase could be neglected.
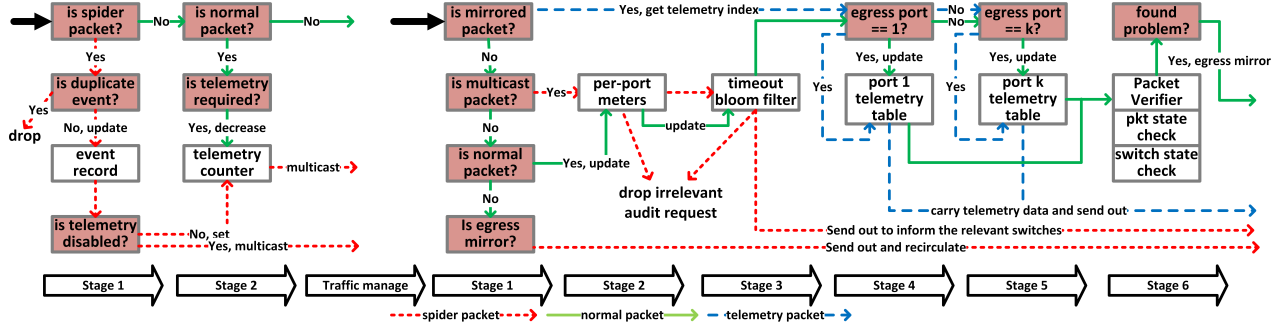
Figure 14: The placement of SpiderMon components on the switch stages

E.2 Cache & Hadoop Workloads

Besides the Web search trace, we also run the same experiments on the Cache trace and Hadoop Trace.

For the Cache trace, most of its flow sizes fall into 1KB to 100KB. Thus, to reach the same link utilization, we have to insert more number flows during the simulation. The results for Cache trace are similar to the Web search trace. The only difference is that all algorithms have improved performance. This is because the flow sizes are very small so that the root-cause traffic (e.g. micro-burst) flow can be easily distinguished from the normal flows; false positive and false negative are reduced.

For the Hadoop trace, most of the flows have less than 10 KB flow size. Similar to the Cache trace, we also increase the number of flows to keep the same link utilization. The overall results for the Hadoop trace are also similar to the Cache trace.

(a) Micro-burst

(b) Priority contention

(c) ECMP imbalance

(d) Loops (solutions except Trumpet all overlap at the top right corner)
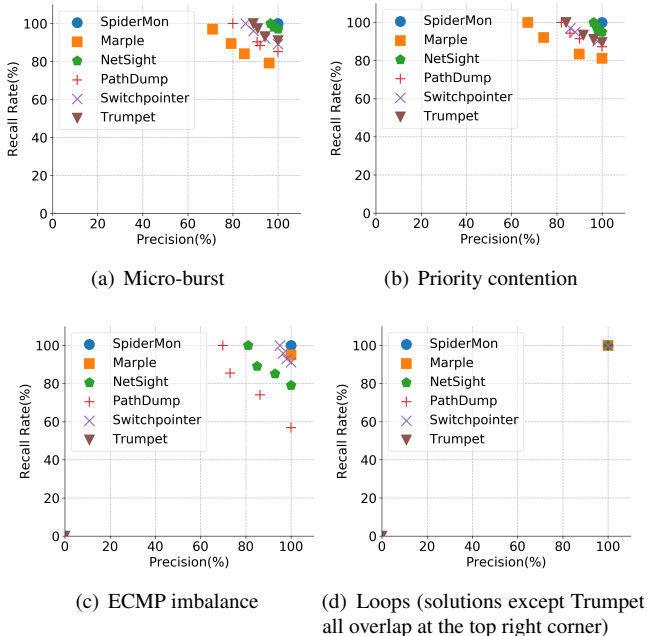
Figure 15: Diagnostic effectiveness with Cache trace

(a) Micro-burst

(b) Priority contention

(c) ECMP imbalance

(d) Loops (solutions except Marple and Trumpet all overlap at the top right corner)
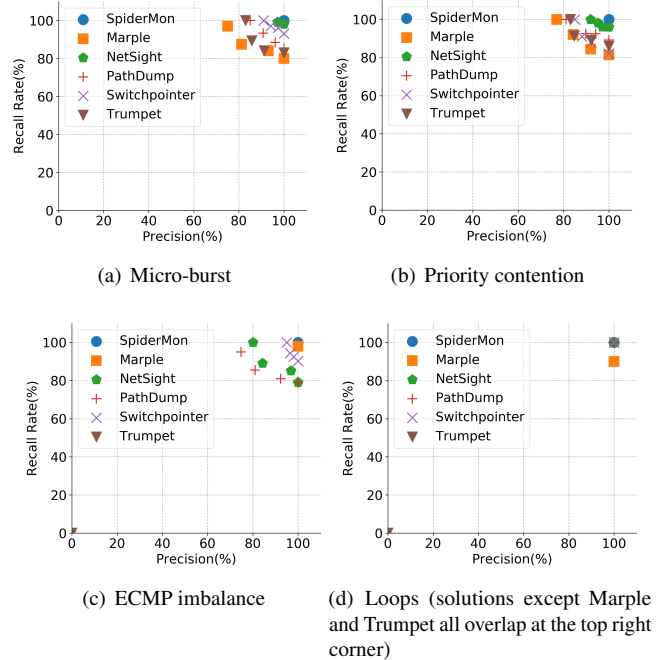
Figure 16: Diagnostic effectiveness with Hadoop trace

F Tunable Parameters for Different Solutions

We vary the following parameters when using those systems to diagnose problems of the four scenarios. The goal is to find the parameter sets with the best precision and recall rate. We do nested iterations over different parameters by fixing some parameters and iterate the other parameters. The parameters are different across systems, and for the same system, the parameters vary according to the scenarios that we are trying to diagnose. The details are shown in Table 2 and Table 3.

G Constructing Signatures for Root Causes

SpiderMon uses both the collected telemetry information and the static network configuration information to recognize the root causes. The telemetry information is collected by SpiderMon, and the configuration information is simply provided

	Micro-burst-related Contention	Priority-related contention
Trumpet	Tolerable per-flow throughput, tolerable end-to-end latency difference, tolerable TCP packet loss	Tolerable per-flow throughput, tolerable end-to-end latency differences, tolerable TCP packet loss
PathDump	Tolerable per-flow throughput, tolerable link utilization	Tolerable per-flow throughput, tolerable link utilization
SwitchPointer	Tolerable per-flow throughput, tolerable link utilization	Tolerable per-flow throughput, tolerable link utilization
NetSight	Related time interval length, tolerable link utilization	Related time interval length, tolerable link utilization, postcard arrival sequences
Marple	Network-wide query lasting time, tolerable per-flow throughput	Network-wide query lasting time, tolerable per-flow throughput
SpiderMon	Maximum allowed flow throughput	/

Table 2: Parameters for micro-burst and priority

	ECMP load imbalance	Loop
Trumpet	/	/
PathDump	Tolerable link utilization, tolerable link utilization imbalance ratio	Maximum header size
SwitchPointer	Tolerable link utilization, tolerable link utilization imbalance ratio	Maximum header size
NetSight	Related time interval length, tolerable link utilization, tolerable link utilization imbalance ratio	/
Marple	Network-wide query lasting time, tolerable link utilization imbalance ratio	Network-wide query lasting time
SpiderMon	Tolerable link utilization imbalance ratio	/

Table 3: Parameters for load imbalance and Loop

by the topology information and routing information, which is known by the operator in advance.

To add a new signature for a new root cause, network operators could simply use the above information to construct their own signatures. Here we provide some telemetry information and static configuration information used in the 4 example signatures in Table 4. This is not an exhaustive list and more information could be added when new signatures are introduced. To construct new signatures, we should know that any signature consists of two parts: 1) the root cause's pattern, like a flow with large throughput for the micro-burst root cause; 2) the relation between the problematic flow and the victim flow, namely, the problematic flow should be one of the main contributors to the victim flow's poor performance. Here we also provide 4 different signatures as examples.

Telemetry Info	Edge weight from flow i to flow j: $E(flow_i, flow_j)$
	Main contributors for a queue: $Contributors(Switch_iPort_j)$
	Flows traveling a switch port: $Flows(Switch_iPort_j)$
	Priority: P(flow)
	Data volume: V(flow)
Config Info	Port mapping in Topology: $Topo(Switch_iPort_j)=Switch_xPort_y$
	Flows belonging to an ECMP group: Flows(group)

Table 4: Selected telemetry information and static configuration information

Micro-bursts. SpiderMon can identify all the main flow-level contributors at different hops along the victim flow's historical path. As shown in Figure 5(a), the micro-burst flows have many wait-for edges with large weights pointing to them-

selves due to a large amount of traffic during the problematic time. For example, for the micro-burst problem, there must exist one micro-burst node *root* which satisfies:

The root cause flow has the same priority as the victim flow:

$$P(victim) = P(root) \qquad (5)$$

The root cause flow has similar edge weight to itself as to other flows:

$$E(root, root) \approx E(victim, root) \qquad (6)$$

The victim flow contends with the root cause flow:

$$\exists m, n, \text{where}$$
$$victim \in Flows(Switch_mPort_n) \qquad (7)$$
$$root \in Contributors(Switch_mPort_n)$$

The larger the weights of $E(root, root)$ and $E(victim, root)$, the more confidence SpiderMon has on determining the micro-burst flow.

Different priorities. For contention between flows with different priorities, SpiderMon checks the priority of the victim flow and the main flow-level contributors. The contributor flows with higher priority compared to the victim flow can be identified as the root causes, as shown in Figure 5(b). The high priority flow *root* should satisfy:

The root cause flow has higher priority than the victim flow:

$$P(victim) < P(root) \qquad (8)$$

The root cause flow has smaller edge weight for the edge pointing to itself than the edge pointing to the victim:

$$E(root, root) < E(victim, root) \qquad (9)$$

The victim flow contends with the high priority flow:

$$\exists m, n, \text{where}$$
$$victim \in Flows(Switch_mPort_n) \qquad (10)$$
$$root \in Contributors(Switch_mPort_n)$$

ECMP load imbalance. For the load imbalance problem displayed in Figure 1(b), SpiderMon will find the flow-level main contributors and check if they are routed by ECMP. Then SpiderMon calculates the ECMP imbalance ratio with the throughput of all flows routed by ECMP rules, using the traffic volume provided by per-flow telemetry data. The problematic ECMP groups can be identified when the calculated ratio is highly imbalanced as in Figure 5(c). Within the problematic ECMP group *ecmp* on Switch $Switch_x$, there must exist one or more flows *root*, which satisfies:

The ECMP traffic split on some switches is not balanced:

$$Throughput(Switch_x Port_y) = \sum V(flow_i),$$
$$\text{where } flow_i \in Flows(Switch_x Port_y) \quad (11)$$

$$\exists x, y, \forall i \neq y,$$
$$Throughput(Switch_x Port_y) \quad (12)$$
$$> Throughput(Switch_x Port_i)$$

The root cause flow is one of the flows from the ECMP port that has larger throughput.

$$root \in Flows(ecmp) \cap Flows(Switch_x Port_y) \quad (13)$$

On another switch, the victim flow contends with the root cause flow:

$$\exists m, n, \text{where}$$
$$victim \in Flows(Switch_m Port_n) \quad (14)$$
$$root \in Contributors(Switch_m Port_n)$$

Transient/persistent loops. For the latency problem caused by transient or persistent loops as shown in Figure 1(c), Spider-Mon searches the port-level contributors along the contributor traffic's path. If the same port is observed twice during the search procedure, all those ports are highly likely to have formed a loop for specific traffic. Furthermore, the flow ID will be checked to further confirm the transient/persistent loop. The formal signature for a flow *root* with a transient/persistent loop can be written as:

$$\text{Exist a port list:} [Switch_{m_0} Port_{n_0}, ..., Switch_{m_k} Port_{n_k}] \quad (15)$$

The port list forms a ring in the topology and the root cause flow routed in a loop on that ring:

$$\forall i,$$
$$Topo(Switch_{m_i} Port_{n_i}) == Switch_{m_{i+1}} Port_{n_{i+1}} \quad (16)$$
$$root \in Flows(Switch_{m_i} Port_{n_i})$$

The victim flow contends with the loop traffic on one of the switches on that ring:

$$\exists j, \text{where } j \in [0, 1, ..., k]$$
$$victim \in Flows(Switch_{m_j} Port_{n_j}) \quad (17)$$
$$root \in Contributors(Switch_{m_j} Port_{n_j})$$

Collie: Finding Performance Anomalies in RDMA Subsystems

Xinhao Kong[1,2]   Yibo Zhu[2]   Huaping Zhou[2]   Zhuo Jiang[2]
Jianxi Ye[2]   Chuanxiong Guo[2]   Danyang Zhuo[1]

[1]Duke University [2]ByteDance Inc.

Abstract

High-speed RDMA networks are getting rapidly adopted in the industry for their low latency and reduced CPU overheads. To verify that RDMA can be used in production, system administrators need to understand the set of application workloads that can potentially trigger abnormal performance behaviors (e.g., unexpected low throughput, PFC pause frame storm). We design and implement Collie, a tool for users to systematically uncover performance anomalies in RDMA subsystems without the need to access hardware internal designs. Instead of individually testing each hardware device (e.g., NIC, memory, PCIe), Collie is holistic, constructing a comprehensive search space for application workloads. Collie then uses simulated annealing to drive RDMA-related performance and diagnostic counters to extreme value regions to find workloads that can trigger performance anomalies. We evaluate Collie on combinations of various RDMA NIC, CPU, and other hardware components. Collie found 15 new performance anomalies. All of them are acknowledged by the hardware vendors. 7 of them are already fixed after we reported them. We also present our experience in using Collie to avoid performance anomalies for an RDMA RPC library and an RDMA distributed machine learning framework.

1 Introduction

Data center applications relentlessly demand low packet latency and high CPU efficiency. That makes Remote Direct Memory Access (RDMA) an appealing solution for cloud providers and other data center operators. Today, many top companies have already adopted RDMA in their data centers [11, 20, 46]. RDMA has been integrated into many application domains, such as graph processing [2, 41], data stores [4, 16], and deep learning [14, 44].

To deploy RDMA in production, i.e., using RoCEv2 for Ethernet-based data center network, we need to make sure that the RDMA network performance can meet our expectations, free of performance anomalies like low throughput and pause frame storm [11, 13, 32, 46]. This is important because applications require high-performance RDMA networks to de-

liver their service-level objectives (SLO). Furthermore, some abnormal behaviors, like pause frame storms, can cause catastrophic consequences including deadlocking the entire data center network [8, 11, 13, 37].

We have encountered the following anomalies in our RoCEv2 production environment:

- A particular application workload's performance of the same RDMA NIC (RNIC) varies substantially on servers with only a slight difference in their PCIe specifications.

- A specific application workload only triggers pause frame storms with certain NUMA settings on a particular RNIC combined with particular server hardware.

- A particular application workload triggers pause frame storms with only a single connection on a particular RNIC from a particular vendor.

Although we collaborate with the most reliable vendors and they have conducted extensive tests on individual devices, the entire RDMA subsystem still has anomalies. The RDMA subsystem consists of RNICs and other server hardware that interacts with the RNICs. Our observation is that most of the anomalies are highly related to the interactions between RNICs and rest of the server hardware. Additional integration tests are thus critical, and we usually conduct these tests on our own because of two reasons. First, vendors cannot access our highly customized hardware, system configurations, and applications. Second, anomalies are too critical for the reliability and performance of the entire data center network, and we cannot completely rely on third parties for testing.

Currently, there are two approaches to conduct tests over the entire subsystem. The first approach is to run simple test benchmarks (e.g., `Perftest` [34]) to conduct basic throughput and latency tests. The second approach is to run a set of representative RDMA applications. Unfortunately, these two approaches are not able to comprehensively uncover RDMA subsystem anomalies. The fundamental problem is that these approaches only test simple or existing workloads. They therefore fail to capture anomalies comprehensively because real

application workloads change over time. In addition, even if an anomaly is found with an application workload, application developers do not know how to modify the workload to avoid the anomaly.

Our goal for this paper is to explore the possibility of **systematic search** for application workloads that can trigger performance anomalies in RDMA subsystems. Finding these anomalies for the vendors can help them improve their hardware and thus improve the reliability and the performance of the entire data center network. Besides, the systematic approach can help developers understand the conditions to trigger such anomalies and how to avoid them by changing application workloads.

To realize this goal, the first question is *how to formally define an anomaly?* Having such a definition is difficult because application performance highly depends on the workload and the hardware. In this paper, we focus on two types of performance anomalies that can be precisely defined: no PFC pause frames if the network is not congested and throughput should be bottlenecked either by bits/second or packets/second as in RNIC specification.

Given this definition, we still need to address three challenges. The first challenge is how to build a comprehensive workload search space. An ideal approach for testing with the entire RDMA subsystem is to exactly modeling each component and then construct the search space. However, this is extremely hard for us, given the black-box nature of RNIC and other hardware components. The second challenge is even after we successfully construct a comprehensive enough search space, how can we search efficiently? The search space is inherently very large because RDMA subsystems are complicated. For example, traffics within an RDMA subsystem can be from/to different memory devices (e.g., main memory and GPU memory) and the transportation setting for a given workload is various (e.g., number and type of connections). Conducting tests blindly in such a large space is inefficient. The third challenge is how to find the complicated triggering conditions of such anomalies? This is important both during the search and after the search. During the search, we need the triggering condition to avoid testing similar application workloads for the same anomaly to speed up the search. After the search, we need to use these conditions to help developers avoid anomalies.

To this end, we design and implement Collie, the first tool to systematically uncover RDMA subsystem performance anomalies, with the following three ideas.

Our first idea is to construct the search space from a developer's perspective. Though the underlying hardware is various and opaque to us, the narrow-waist RDMA programming abstractions (i.e., *verbs*) are clearly defined and stable. All application workloads can be interpreted as a combination of *verbs* operations. We carefully analyze the standard *verbs* library and the design decisions developers are allowed to make (the request pattern, how RDMA buffers are allocated, etc.). Moreover, to cover the entire RDMA subsystem, we analyze all the potential data flows within a given server configuration. In this way, Collie constructs a comprehensive search space for application workloads in the domain of RDMA subsystem, including the host of the network traffic (e.g., GPU connected to a different PCIe bridge from the RNIC, DRAM from a different CPU socket), message sizes, number of connections, and memory region configurations.

Our second idea is that we can use two sets of counters to guide the search. The first set is the performance counters (e.g., bits per second), which are provided by all commodity RNICs and other hardware components. In addition, modern commodity RNICs and other hardware components provide diagnostic counters (e.g., PCIe backpressure). Diagnostic counters are mapped to particular unexpected events that happen to the hardware components. These counters are currently only used for debugging and monitoring purposes. Collie uses search algorithms based on simulated annealing to maximize/minimize counter values to uncover anomalies.

Our third idea is to find the minimal area in the search space that covers the found anomalies. We call this area (i.e., the conditions to trigger the anomaly) the minimal feature set (MFS). Collie includes a MFS algorithm to test each feature that an anomaly has (e.g., number of connections) and generate the necessary conditions set. With the MFS algorithm, Collie can further improve search efficiency by avoiding redundant tests of the same area. Also, finding the triggering conditions of an anomaly allows developers to avoid the anomaly by breaking one of the provided conditions.

We evaluate Collie on 8 different RDMA subsystems, including 6 types of RNICs from NVIDIA Mellanox and Broadcom, with speeds between 25 Gbps and 200 Gbps. Before we build Collie, we already know 3 existing performance anomalies by testing with existing RDMA applications. Collie successfully reproduces all of them and has found 15 new anomalies. We report these anomalies to the vendors, and all of them are acknowledged. 7 of them are already fixed by firmware upgrade or detailed configuration following our vendors' instructions. We also describe our experience in using Collie to guide an RDMA RPC library and an RDMA distributed machine learning framework to avoid these anomalies. These experiences show Collie can help data center operators to uncover anomalies and assist RDMA application developers to implement better applications.

This work makes the following contributions:

- We design a developer-oriented approach to systematically construct a search space of application workloads to find performance anomalies in RDMA subsystems.

- We propose the first work to leverage hardware counters to guide the search for performance anomalies. These counters do not have proprietary hardware knowledge. This makes Collie general and useful for all types of RDMA subsystems.
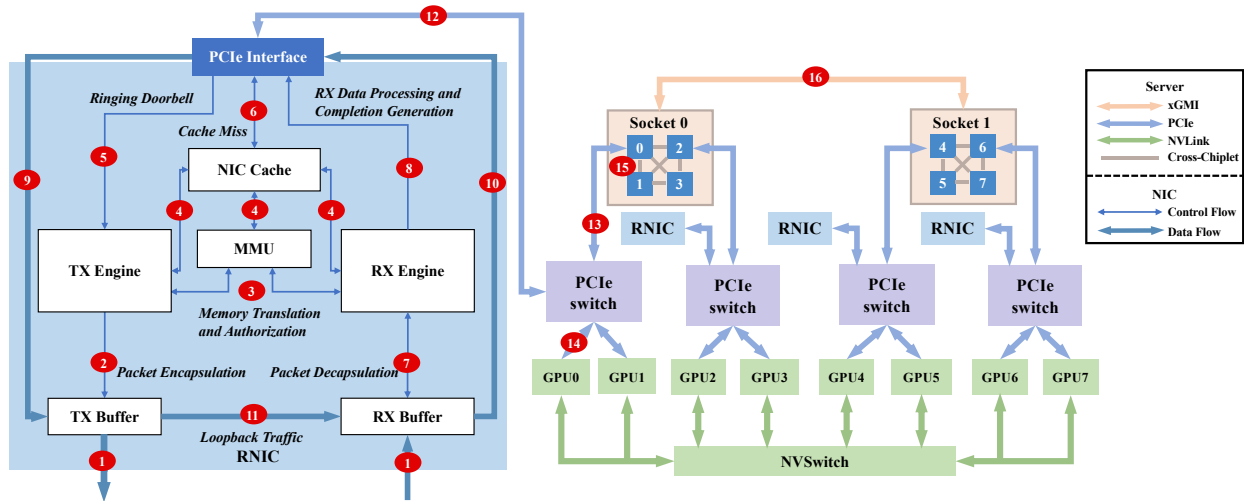
Figure 1: An example of an RDMA subsystem (RNIC internal design and its deployment environment in a server). Red circles mean potential performance bottlenecks that can trigger performance anomalies.

- We develop a simulated annealing based search algorithm and MFS algorithm. These algorithms speed up search and help developers avoid anomalies.

- We implement Collie, the first tool to help data center operators to uncover and avoid RDMA subsystem performance anomalies. Collie has found 18 anomalies (3 known ones and 15 new ones). We present these anomalies, their mitigation strategies, and their implications.

2 Background

2.1 RDMA Subsystem Performance Anomalies

RDMA is increasingly deployed in data centers for applications to achieve high throughput and high CPU efficiency. An application process can directly communicate through an RNIC with a remote process without involving either side's CPUs. RDMA requires a lossless network to achieve high performance. The default technology to deploy RDMA for Ethernet-based data centers is RoCEv2 [11,46]. It relies on Priority-based Flow Control (PFC) [35] mechanism to guarantee a lossless network: once an ingress queue length exceeds a threshold, the switch/NIC sends out a PFC pause frame to the upstream egress queue, asking the egress queue to pause for a duration to avoid ingress queue overflow.

RDMA subsystem performance does not always meet user expectations and can have severe performance anomaly. According to our production experience, specific application workloads can trigger hardware bottlenecks of a particular type of RDMA subsystem and cause the entire subsystem performance to drop drastically. Applications of the same subsystem will be affected (e.g., throughput drop) and miss the service level agreement. Worse still, an anomalous RDMA subsystem can send out a large amount of PFC pause frames, which pauses the priority queue of the corresponding switch

port and may threaten the entire data center network, such as causing head-of-line blocking and PFC deadlocks [11,13,28]

Though the vendors of RNICs and other hardware components (e.g., GPU, motherboard) have conducted extensive tests on their products, we still find many anomalies in our RoCEv2 production environment. The fundamental reason is that RDMA performance is highly related to the entire RDMA subsystem, consisting of both RNIC internals and other hardware components. Figure 1 shows the complexity of an RDMA subsystem. **This figure is based on public resources [24,32,42] and does not expose proprietary information. Our conversation with Mellanox indicates that a real RNIC is much more complex than our figure shows.** To the best of our understanding, an RNIC has at least 6 components: (1) a *TX engine* that receives doorbells (a signal mechanism for the server to notify RNIC to send a request), fetches and processes requests, and initiates transmission; (2) an *MMU* that translates the virtual address to physical address for RDMA memory regions; (3) an SRAM-based *NIC cache* that caches per-connection metadata and memory translation table; (4) a *RX engine* that processes incoming data and generates completion to notify server; (5)(6) *buffers* that hold packets to transmit and received packets. An RNIC is connected to a server via PCIe. The server has two CPU sockets and each CPU socket has four CPU chiplets (Only AMD CPUs and new-generation Intel CPUs have cross-chiplet communication, otherwise all the cores inside a CPU socket share the last-level cache.) RNICs and GPUs are all connected to PCIe switches.

There are many potential performance bottlenecks inside the RNIC and between the RNIC and other hardware components within the RDMA subsystem. We use red circles to show such potential bottlenecks (in Figure 1). When these bottlenecks are triggered, the network performance may drop and the RNIC can even send out pause frames to reduce

the amount of traffic going through the RNIC. We find that many anomalies only occur when multiple bottlenecks or the bottlenecks between different components are triggered. For example, when the RNIC receives a packet, it will store the packet in RX buffer, process the packet (circle 7), and finally DMA the content to main memory or GPU memory (circles 10, 12, 13 or circles 10, 12, 14). Normally, the RX buffer won't accumulate much because the PCIe bandwidth is larger than RNIC's line rate (circle 1). However, once there exists loopback traffic (e.g., the client and server are collocated on the same host), the loopback traffic (circle 11) may drain the PCIe bandwidth and cause RX buffer accumulation. It depends on both the RNIC and the PCIe slot. The worst consequence is that the RNIC keeps sending a large amount of PFC pause frame and threatens the entire data center network. Vendors' individual tests are not able to uncover this anomaly because it depends on the combination of circles 1, 11, 12 (even more) from different components. Further, data center operators like us may use highly customized hardware or specific system configurations that are not accessible to vendors. This makes it necessary and crucial for us to conduct our own independent tests before deploying RDMA hardware in production, especially for anomalies that can potentially generate pause frame storms.

2.2 Existing Approaches

Data center operators' tests are integration tests: instead of testing individual hardware components, these tests focus on the performance of the entire RDMA subsystems. There are two existing approaches. The first approach is to run a set of test traffic, such as `Perftest` [34] and `OSU micro-benchmarks` [33]. The second approach is to run a representative set of real applications. However, these two approaches can not uncover RDMA subsystem performance anomalies comprehensively. For example, we deploy 200 Gbps RNICs in our clusters to support a performance-critical distributed machine learning framework. We test the machine learning framework on the cluster of these RNICs, and there is no performance anomaly found. We also have done extensive testing both with synthetic testing workloads and other real applications before deployment. However, months after deployment, our developers find that the performance of the framework has reduced significantly, even worse than just using 100 Gbps RNICs. At the same time, a substantial amount of pause frames are generated from these 200 Gbps RNICs. This is strange because pause frames usually appear with hundreds of connections that trigger congestion, but our machine learning framework only creates a few connections between each server pair. We stopped the machine learning framework and ran our performance tests again, and everything is normal. After several weeks of careful debugging, we finally realize that the case only happens when the application (1) use one-sided RDMA operations with Reliable Connection, (2) has bidirectional traffic, (3)
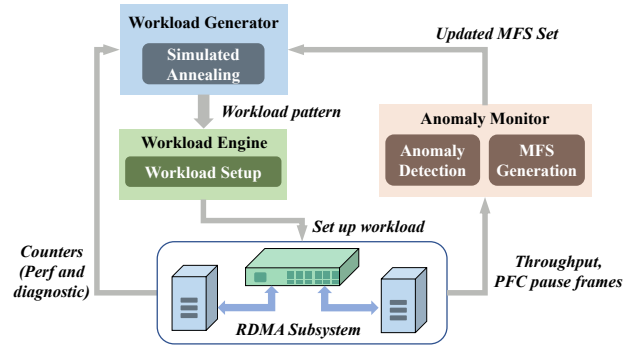
Figure 2: System Overview. The workload engine sets up RDMA traffic. The anomaly monitor detects performance anomalies and their minimal feature sets. The workload generator fetches hardware counters and decides the workload pattern to test.

uses a particular workload including a mixture of small and large messages, (4) with 200 Gbps RNIC on particular AMD servers. We find that the developers for our machine learning framework slightly modified their code after passing our application tests. The new workload contains messages of mixed lengths (i.e., a large message followed by a small message followed by a large message in bidirectional traffic), which triggers a performance bottleneck between the RNIC and the PCIe switch. This is not a problem with 100 Gbps RNICs from the same vendor or on other types of servers.

The fundamental reason why current approaches fail to uncover such anomalies is that they only test existing workloads and inherently are not able to capture anomalies triggered by unknown workloads. However, real application workloads are various and will change over time. Besides, even current approaches have found such anomalies, it is hard and time-consuming to locate the triggering conditions. Capturing the triggering conditions of performance anomalies allows data center operators to work with vendors to fix potential hardware/firmware bugs, and improve the reliability and performance of the data center network. When fixes to the anomalies are not immediately available (e.g., firmware upgrade, hardware replacement), application developers can build high-performance RDMA applications by avoiding workload that can trigger anomalies.

3 Overview

We build Collie, the first tool to help data center operators systematically search for application workloads that can trigger performance anomalies.

The first question we need to answer is *how to define an anomaly?* Unfortunately, today there does not exist such a definition. Having such a definition is fundamentally hard because application performance (e.g., latency) can be highly dependent on the workload and the hardware. Instead of trying to capture the entire set of anomalous behaviors, we focus on two types of performance anomalies that are of great importance in production environment and can be precisely defined: when applications keep injecting RDMA traffic into

the network, (1) no PFC pause frames should be generated if the network is not congested; (2) throughput should be bottlenecked either by total bits/second or total packets/second as in RNIC specifications. The first definition ensures that an RDMA subsystem will not threaten the entire data center network and the second ensures that an RDMA subsystem's capability matches user expectation. [1] We discussed this definition with several hardware vendors, and they all agree with our definition. Even though some anomalies may be due to system limitations rather than bugs, it is also important for both vendors and us to be aware of them. We report all the anomalies we found using this definition to the hardware vendors, and they acknowledged all the reported anomalies. We believe that this definition naturally matches the application developer's mental model of RDMA and thus allows developers to roughly estimate the network performance.

Given this definition of anomaly, we still need to overcome three major research challenges.

Challenge #1: How to design a comprehensive workloads search space for a given RDMA subsystem? An ideal solution is to carefully analyze and modeling the entire RDMA subsystem, and then construct the search space from the perspective of hardware. This complete white-box approach allows us to test all bottlenecks and the combinations of them givens an RDMA subsystem. However, it is impractical for data center operators like us due to the black-box nature of RNICs and other hardware components. Our key observation is that though the components of RDMA subsystems are black boxes and there are diverse RDMA applications, the abstractions between the hardware and applications are clearly defined and stable. All application workloads are essentially composed of a series of basic *verbs* operations, a *narrow waist* of the RDMA programming. With this observation, we carefully analyze this RDMA programming abstraction and design a general search space (§4).

Challenge #2: How to search efficiently? Due to the complexity of RDMA subsystems and the variety of workloads, the size of search space is very large. Unfortunately, none of existing heuristic search algorithms can be directly applied due to the lack of a search signal (e.g., direction for the next workload to test). We observe that there are two sets of counters commodity RDMA subsystem provide can be leveraged to guide the search. The first set is known as performance counters. For example, all modern RNIC provide the counter of bits sent per second for monitoring purpose. The second is known as diagnostic counters. Modern RNICs and other hardware components expose diagnostic counters for debugging purpose (e.g., the counter indicates PCIe backpressure and NIC internal cache miss) [22, 23]. Diagnostic counters

are more informative. For example, when some bottlenecks of the RDMA subsystem are triggered, the performance may not drop while the corresponding diagnostic counter has increased. However, using diagnostic counters typically requires vendor's assistance, and the number of diagnostic counters customers can access depends on vendors. For Collie to be general, we use both performance counters and optionally diagnostic counters as search signals. We conduct the efficient search by using a simulated annealing based algorithm to drive these counters to extreme value regions (§5.1).

Challenge #3: How to find the set of conditions to trigger anomalies? Some anomalies are complicated and only occur when many features co-exist, such as a certain type of transportation, particular message pattern, lots of connections, and specific batching operations. We invent a minimal feature set (MFS) algorithm to detect each factor's contribution to the uncovered anomaly and construct the necessary conditions set. To search efficiently, we use MFS to avoid testing similar workloads that map to the same anomalies. After the search, developers use the MFS to understand the triggering conditions for each anomaly and bypass them accordingly when the fixes are temporarily unavailable (§5.2).

Figure 2 shows our system design. Collie consists of three core components: (1) a workload engine that conducts experiments on RDMA subsystems by setting up RDMA traffic; (2) an anomaly monitor that detects performance anomaly and MFS to reproduce the observed anomaly; and (3) a workload generator that decides the next workload pattern to experiment based on the counters collected in the RDMA subsystem and the current search space. All the experiments Collie does are on the RDMA subsystem with two servers with RNICs, connected with a commodity switch.

4 Search Space and Workload Engine

There are two types of factors that can affect an RDMA subsystem performance in deployment. First, we need to consider the application workloads. These include host topology (i.e., where does traffic come from inside a server), how many memory regions the application registered, what transport applications choose to use, and the message patterns. Second, we need to consider the network behavior, for example, congestion on switch and packet loss rate. Currently, our paper focuses on constructing a comprehensive search space for the first factor. For the network behavior, we consider a simplified environment: two RNICs connected by a single switch, and there is no packet drop on the switch. Collie can be easily generalized to test more complicated environments.

We take the bottom-up approach to construct a comprehensive search space for various application workloads. We decompose application workloads into combinations of basic RDMA operations and construct the search space based on these combinations. Figure 3 shows the key abstractions and operations of RDMA programming. These are only high-level software abstractions exposed by standard *verbs* API and we

---

[1]We do not use latency as a metric to define anomalies. The only latency specification for RNICs is the latency under zero load. We did not observe any anomaly in this way, probably because the RNIC is not stressed. However, when RNIC is stressed, it is hard to accurately define the correctness of latency or tail latency due to queuing delay.
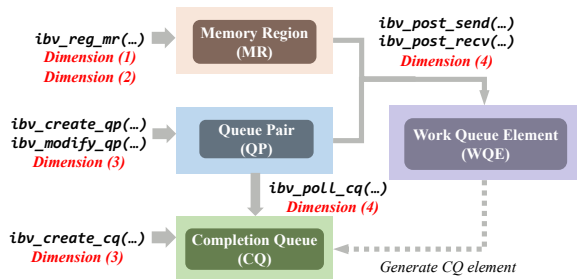
Figure 3: RDMA programming abstractions

do not need to know how these high-level abstractions are implemented in the RNIC. In this way, the search space is more comprehensive and general because it does not rely on either extra proprietary RDMA subsystem hardware knowledge or specific application features. In addition, the combinations of *verbs* operations are inherently able to describe workloads of both single application and co-existing scenarios.

We examine the RDMA programming model at first and extract four search dimensions that jointly describe the application workloads of the entire subsystem. To send a message through RDMA networks, applications first need to register a set of memory regions (MRs), using `ibv_reg_mr`. Once registered, an RNIC has the right to directly access these MRs without CPU involvement. Second, applications create a set of queue pairs (i.e., connections in traditional networking terminology), using `ibv_create_qp` and `ibv_modify_qp`. Applications need to choose a transport type for each queue-pair (QP). There are three standard types of QPs: Reliable Connection (RC), Unreliable Connection (UC), and Unreliable Datagram (UD). Applications can use `ibv_post_send` or `ibv_post_recv` to post a list of Work Queue Element (WQE). Each WQE represents a work request and has a scatter-gather (SG) list. Each SG list contains a list of entries and each entry designates a contiguous memory region that is within the registered memory regions. A WQE can notify the RNIC to READ/WRITE remote memory (1-sided operation) or SEND/RECV local memory to/from a remote server (2-sided operation). To know that a WQE is complete, the application can register a completion queue (CQ) using `ibv_create_cq`, and the application can call `ibv_poll_cq` to poll on the CQ to get completion queue elements (CQE). Given this RDMA programming model, we extract the following search dimensions.

Dimension 1. Host Topology. Host topology describes how traffic flows to/from an RNIC to/from other server hardware components. Individual component tests are hard to cover this dimension while the topology has a huge impact on RDMA subsystem performance. For example, traffic can be from NUMA-affinitive DRAM or from a GPU that needs to traverse both PCIe and SMP interconnect between NUMA nodes. The latter will have a longer data path and therefore higher average DMA latency. This will trigger PCIe backpressure to the RNIC and may induce performance anomalies under some

specific application workloads. We list all accessible memory devices for this dimension.

Dimension 2. Memory Allocation Settings. Traditional RDMA testing is not comprehensive for this dimension, while the memory allocation settings are crucial for RDMA subsystem performance testing. First, the number of MRs affects RDMA subsystem performance because RNIC has an MMU that translates virtual addresses of memory regions to DMA-capable physical addresses and handles memory protection (e.g., authorization). RNIC only caches a fixed size of entries of the memory address translation table. If too many MRs are registered, it is then likely that the RNIC encounters cache miss and needs to access memory address translation tables on server DRAM via extra PCIe operations. These interactions have an impact on the performance. Second, MRs can have different sizes. This also affects RDMA performance because the size also affects the number of translation table entries. Moreover, many RNICs use Intel Data Direct IO (DDIO) to directly access the CPU's last-level cache. If the access range of an MR is large, it can cause severe cache misses in the CPU's last-level cache [3]. This dimension is bounded because we can set a reasonable upper bound on the number of MRs (200K), and the MR size is bounded by the total amount of memory that can be registered (pinned) in the physical server.

Dimension 3. Transport Setting. Transportation setting is crucial for RNIC performance, and this is well known in the research community [15,17,27]. We use the following factors to compose the transport setting: (1) QP type (RC, UC, UD), (2) the number of QPs, (3) the opcode type (SEND/RECV, WRITE, READ), and (4) the usage of SG and WQE. Different QP type with different opcode creates different pressure for the RNIC. For example, UD does not require ACK for each packet, which lessens the RNIC packet processing pressure. However, the SEND/RECV requires pre-posted receive buffers, which puts more pressure on the RNIC cache. The number of QPs also has a great impact on RNIC performance because of the limited RNIC cache. This is known as the scalability problem [3,15,32]. How SG list and WQE affect RNIC performance is a bit tricky. RNICs have to consume extra PCIe bandwidth to fetch WQE from the host DRAM [17]. The PCIe bandwidth consumed by WQE becomes substantial under some particular application workloads and can even be the performance bottleneck. We enumerate all the transport types and the opcodes (e.g., RC WRITE, UD SEND). It is practical and reasonable to set an upper bound (e.g., 20K) for the number of QPs because data center operators will hardly set up more connections. The SG list and WQE can be parameterized by this formula: $\sum_{i=1}^{n} m_i = k$, where k denotes the number of messages to send, n denotes the number of WQE and m_i denotes the number of SG elements within the ith WQE.

Dimension 4. Message Pattern. Existing RDMA testing approaches lack flexibility and comprehensiveness, es-

pecially for this dimension. `Perftest` [34] only repeatedly send messages of a fixed size and other collective communication benchmarks (e.g., `OSU benchmark` [33]) test RDMA similarly. These simple benchmark traffic are inadequate for RDMA subsystem testing because they ignore the interaction among different requests (i.e., WQE) in a sequence.

Our ideal goal is to construct this dimension that can represent any application message pattern. However, it is impractical because application traffic can be very different and the interaction among different requests is unknown given the black-box nature of RNIC. We therefore construct this dimension in the following way. We build a request vector with n elements, where each element describe the request attribute (e.g., size of the message to send). We assume that the 1st request affects the 2nd, the 3rd, ..., the nth requests but won't affect the request after the nth. The larger n we set the larger search space we can cover, but we also need to consume more time. This kind of trade-off is similar to the approach when testing file systems [21], where researchers test fixed-length file system operation sequences. Modern RNIC has limited Processing Units (PU) and pipeline stages [39], restricting the number of outstanding requests an RNIC can process. We thus set n to be the product of the number of PUs and the pipeline stages. We discretize request size into multiple discrete value regions based on MTU and the burst size of the RNIC. The RNIC splits a long request into multiple bursts and processes each burst at one time to avoid Head-of-Line (HoL) blocking. The granularity can be easily modified. With more search time, we can discretize request size in a more fine-grained way. Message inter-arrival time is usually considered as a parameter for application workloads. However, adding the inter-arrival time will substantially extend our search space, so we temporarily only consider the pattern without such inter-arrival time.

Workload engine. We build a flexible workload engine to conduct tests in our search space. Compared to traditional traffic generation tools, e.g., `Perftest`[2], our workload engine is more flexible and has a holistic view. It can send and receive traffic with particular pre-defined patterns (e.g., a large WRITE request followed by a small SEND request). Besides, it supports various memory and transport settings, which can test the entire subsystem holistically. To test with a point in our search space, Collie first translates a test point's settings into a set of input parameters of the workload engine. For example, the setting of dimension 1 and 2 are translated into memory allocation parameters (i.e., which GPU or NUMA DRAM to use and how many MR to register) of the engine. Then, the workload engine will take these input parameters to set up connections and generate traffic.

---

[2]Existing tools, e.g., `Perftest`, are arguably not designed for this type of testing. They are performance benchmark tools. However, we are not aware of any other tools can that test RDMA subsystems.

Algorithm 1 Search for Performance Anomalies

Input: S: initial anomaly set; T_0: a high enough initial temperature; T_{min}: the lowest limit of temperature; n: the number of times SA runs for a certain temperature;

Output: S: An updated anomaly set;

1: $P_{old}, M_{old} = MeasureRandomPoint()$; pick a random point, setup traffic and collect metrics as M
2: **while** $T > T_{min}$ **do**
3:     **for** $i = 0; i < n; i++$ **do**
4:         mutate P_{old} for a new application workload P_{new};
5:         **if** $MatchMFS(S, P_{new})$ **then** continue;
6:         $M_{new} = MeasurePoint(P_{new})$;
7:         $\Delta E = CompareMetric(M_{new}, M_{old})$;
8:         **if** $\Delta E < 0$ **then**
9:             $P_{old} = P_{new}$
10:         **else**
11:             the probability $prob = exp(-\Delta E / T_{(i)})$;
12:             **if** $rand(0,1) < prob$ **then** $P_{old} = P_{new}$;
13:         **end if**
14:         **if** $IsAnomaly(M_{new})$ **then**
15:             $new\_mfs = ConstructMFS(P_{new})$;
16:             Put new_mfs into S;
17:             $P_{old}, M_{old} = MeasureRandomPoint()$; pick another random point when a new anomaly is found
18:         **end if**
19:     **end for**
20:     $T = T * \alpha$; where α is decay factor
21: **end while**
22: **return** S

5 Search for Performance Anomalies

The total size of our search space (i.e., the combination of parameters) is on the order of 10^{36}. Each experiment we do requires 20-60 seconds, mostly depending on the number of QPs to create and the number of MRs to register. This means we cannot exhaust the search space. One naive approach is to generate random input in the search space. This approach is already much better than existing tests because the design of our search space is more comprehensive than that in existing tools (§7). However, similar to typical black-box fuzz testing on software, random inputs can only find few anomalies and cannot efficiently uncover complicated anomalies that require multiple conditions to hold simultaneously.

5.1 Workloads Generation

We leverage two types of counters to guide the search. The high-level approach is to use an optimization algorithm to drive counters to extreme value regions by keeping mutating the test workloads. For performance counters, we drive the counters to low-value regions. For diagnostics counters (which map to unexpected events), we drive the counters to high-value regions.

Our algorithm is based on simulated annealing (SA). SA is a probabilistic algorithm to find the global minimum of a given function. The idea is to keep mutating the input in the direction of minimizing a given function. SA calls the func-

tion value energy. To avoid getting stuck at a local minimum, SA maintains a temperature value. At the beginning of the algorithm, the temperature is high and SA allows mutating input in the direction of increasing the energy. As temperature decreases during the search, SA is less likely to move the input in the direction of increasing the energy. Finally, when the temperature is below a certain threshold, every mutation of the input must decrease energy. SA finishes when there is no way to mutate the input to make the energy lower.

Algorithm 1 shows our algorithm that is based on SA. We maintain a list of performance anomalies. Each anomaly is an MFS (e.g., an area in the search space) that contains workloads to reproduce the performance anomaly. The search starts from a random workload in the search space, and our algorithm measures the counter values. In each iteration of SA, we mutate the workload in one of our search dimensions (line 4). We test whether the new workload causes a performance anomaly with our anomaly monitor. If so, we run our MFS algorithm to determine the entire area in the search space that belongs to this anomaly. We add the new anomaly to the set and change the current workload to a random one. If the new workload does not trigger a performance anomaly, we measure the point by comparing counter values and decide whether to move the current workload to the new one. We always skip workloads that belong to an existing performance anomaly for efficient search.

Our algorithm extends the standard SA algorithm in several important ways to adapt it for our context. First, we compute the energy in the following way: assuming the counter value changes from A to B, we set the different in energy (ΔE) to be $\frac{B-A}{A}$ for performance counters and $\frac{A-B}{B}$ for diagnostic counters because we are minimizing performance counters and maximizing diagnostic counters to trigger potential anomalies. This also allows us to avoid value region problem (e.g., the value regions of diagnostic counters are sometimes opaque). Second, we do not require SA algorithm to find the actual global optimum because we care about all potential anomalies. We therefore always set a more relaxed temperature and α that enable the algorithm to jump out of a certain stage even when it has already run lots of iterations. In addition, we maintain a set of performance anomalies (i.e., MFS). When mutating the point, we compare the mutated point with our existing MFS (line 5). Each MFS contains a list of parameters ranges. If the mutated point matches all parameters ranges of an MFS (i.e., the parameter value of this point is in the MFS's range), we claim this point belongs to the MFS and skip testing it. This ensures that the future search does not redundantly test workload already covered by the existing set of anomalies.

5.2 Anomaly Monitor

Our anomaly monitor detects performance anomalies and computes the MFS of them.

Anomaly Detection Condition. We use two conditions to detect anomalies. First, if any pause frame is generated. Here we use a metric called *pause duration ratio*. If the pause duration ratio is 1%, this means for every second, transmission is paused by 10 ms. We set our threshold to be 0.1%. The reason is our experiment platform only has two servers and our switch that connects the servers support line rate traffic, so there is no network congestion to begin with. We set the threshold to be above 0, because RNIC may generate a few pause frames when the memory bus or PCIe bus is busy temporarily, especially when connections are just set up. Second, each RNIC has its maximum bits per second and maximum packets per second in its specification that can be easily verified by running simple benchmarks. Without performance anomalies, network traffic should be restricted by either one of these upper bounds. If a workload's throughput (in terms of both metrics) is 20% lower than the upper bounds, it means that the performance is likely to be restricted by some other bottlenecks of the RDMA subsystem. Collie reports this and runs the MFS algorithm below.

Minimal Feature Set (MFS). After we detect an anomalous workload, we need to know what features of this workload actually trigger the anomaly. For example, if we currently find a new anomaly that has 5 features. It may be the case that 3 features are already sufficient to reproduce this anomaly. One approach is to use machine learning based algorithms to generate decision trees or deep neural networks to locate the area in the search space for the anomaly. However, machine learning approaches usually require much more training data and thus many more hardware experiments.

We instead use a heuristic approach. Since we only have 4 search dimensions with few factors, we just do a few tests on each dimension to determine whether a factor belongs to the MFS. For example, if our search algorithm finds a certain workload using UD can cause a performance anomaly. We test whether the same workload with RC and UC can cause performance anomalies. If not, UD belongs to the MFS because it is necessary to reproduce the anomaly. To determine the MFS of a dimension that is continuous (e.g., number of connections), we discretize them manually into a set of value regions and test each of them. Finer-granularity discretization is acceptable because MFS algorithm only runs when uncovering a new anomaly and the number of anomalies is relatively small compared to the entire search space.

We report all the anomalies to RNIC vendors and we can wait for their fixes. Unfortunately, the solutions to these anomalies are case by case. Some anomalies require vendors to spend a substantial amount of time on coming up with solutions and the solutions may not be applicable for data center operators immediately, such as hardware replacement. Hence, developers need to avoid such anomalies instead of waiting for a fix. Collie provides MFS to help developers avoid such anomalies by changing application workload to break the conditions in the MFS.

MFS helps developers to avoid anomalies in two areas.

Type	RNIC	Speed	CPU	PCIe	NPS	Memory	GPU	BIOS	Kernel
A	CX-5 DX	25 Gbps	Intel(R) Xeon(R) CPU 1	3.0 x 16	1	128 GB	-	INSYDE	4.19
B	CX-5 DX	100 Gbps	Intel(R) Xeon(R) CPU 2	3.0 x 16	1	768 GB	-	AMI	4.14
C	CX-5 DX	100 Gbps	Intel(R) Xeon(R) CPU 2	3.0 x 16	1	384 GB	V100	AMI	5.4
D	CX-6 DX	100 Gbps	Intel(R) Xeon(R) CPU 2	3.0 x 16	1	768 GB	-	AMI	4.14
E	CX-6 DX	200 Gbps	AMD EPYC CPU 1	4.0 x 16	1	2 TB	A100	AMI	5.4
F	CX-6 DX	200 Gbps	Intel(R) Xeon(R) CPU 3	4.0 x 16	1	2 TB	A100	AMI	5.4
G	CX-6 VPI	200 Gbps	AMD EPYC CPU 1	4.0 x 16	2	2 TB	-	AMI	5.4
H	P2100G	100 Gbps	Intel(R) Xeon(R) CPU 2	3.0 x 16	1	384 GB	-	AMI	5.4

Table 1: Testbed RDMA subsystems configurations. We use numbers in the name of concrete CPU types for confidentiality.

The first one is anomaly prevention. Before an application is implemented, Collie lets developers restrict the search space using their knowledge of their applications to represent all the possible workloads. Then, Collie outputs whether the restricted search space contains performance anomalies. If not, assuming the developers' understanding of their applications is correct, the application won't encounter any performance anomaly found by Collie. The second one is debugging. When an existing application unfortunately encounters anomalies, we can run Collie on the RDMA subsystem and generate all the MFS. Comparing the application with the generated MFS, Collie provides several suggestions that help to break the triggering conditions. We present two real cases to show how MFS helps developers in §7.3.

One caveat of our approach is that we are not able to know the root causes of these anomalies given the black-box nature of the RNICs and other hardware components in the RDMA subsystem. This means it may be the case that multiple MFS are actually due to the same anomaly (i.e., the same hardware bug). This is acceptable because the goal of MFS is to accelerate the search algorithm by eliminating redundant test cases and help developers understand what features of the workloads can trigger the anomaly. We anyway need to report all the anomalies (i.e., all the MFS) we found to the vendors and that is also the best we can do given the RNIC black-box hardware nature.

6 Implementation

The workload generator and the anomaly monitor are written in ~2100 lines of Python. The workload engine is implemented with ~2000 lines of C/C++. We directly use monitor tools from vendors to collect hardware counters (both performance and diagnostic counters) from the RDMA subsystem.

The workload engine is implemented with the *verbs* API and *rdma-core-34.0* libraries [38]. In deployment, the Mellanox RNIC uses mlx5 driver (OFED 5.2-1.0.4.0) and the Broadcom RNIC uses bnxt driver (1.10.1.216.2.89.0). The workload engine set up connections by TCP out-of-band transmission. When all connections are set up, the engine starts to generate workload.

The anomaly monitor collects primary metrics, such as throughput and pause frame duration, four times per iteration. It first decides whether the traffic is stable and then compares the primary metrics (e.g., bits per second, packets per second)

with the pre-defined thresholds.

The workload generator collects counters using monitors provided by vendors. These monitors provide counters every second. Collie fetches these counters four times per iteration and uses the average results.

7 Evaluation and Experience

We evaluate Collie on 8 different RDMA subsystems. Table 1 shows the hardware and related configurations. We use the same anomaly detect conditions as described in §5.2

7.1 Performance Anomalies Found

Before we build Collie, we already know 3 existing anomalies. Collie can find all the existing ones and find 15 new anomalies. All of them are reported to our vendors and are acknowledged by them. Table 2 shows the 18 anomalies. We only present those found on subsystem F and H because anomalies found on other subsystems are subsets of those found on F. Appendix A provides details about these anomalies, including the exact workload, as well as the explanations and solutions from vendors. Here we choose two tricky anomalies to show the importance of Collie's systematic search.

Anomaly #4: Bidirectional RC READ with large WQE batch size, long SG list, and a few connections causes PFC pause frames. Our vendors have successfully reproduced this anomaly in their environment using Collie's traffic generator and acknowledged it, but currently there is no fix. This anomaly cannot be found by existing approaches such as using `Perftest` to generate workloads, because `Perftest` does not support flexible WQE and SG list batching strategies. Though `Perftest` is not designed for this purpose, it is the prevalent tool to uncover performance anomalies. To the best of our knowledge, we don't see any other state-of-the-art work address this problem, which also shows that Collie is the first work to fill this vacancy.

Anomaly #10: Bidirectional RC WRITE with large WQE batch size, particular message pattern, and a few connections causes PFC pause frames. This anomaly is not captured by existing approaches (e.g., running current applications) but we successfully reproduce it by slightly modifying our production RDMA RPC library: when users call the library to send a message, it will try to send as many messages as possible in a WQE batch. The batch size is highly dependent on the timeout value. If the application is throughput sensitive rather than

#	RNIC	Direc.	Transport	MTU	WQE	SGE	WQ depth	Message Pattern	# of QPs	Symptom
#1	CX-6	-	UD SEND	-	≥64	-	≥ 256	-	-	pause frame
#2	CX-6	-	UD SEND	-	≤8	-	≥1024	≤1KB	≥≈16	low throup.
#3	CX-6	-	RC READ	1K	-	-	-	≥16KB	-	pause frame
#4	CX-6	Bi-	RC READ	-	≥32	≥4	-	-	≥≈160	pause frame
#5	CX-6	-	RC SEND	1K	≥64	-	≥1024	≥2KB and ≤8KB	-	pause frame
#6	CX-6	-	RC SEND	1K	≤16	≥2	≥1024	≤1KB	≥≈32	low throup.
#7	CX-6	-	RC WRITE	-	No	-	-	≤1KB and ≥≈12K MRs	-	low throup.
#8	CX-6	-	RC WRITE	-	No	-	≤16	≤1KB	≥≈500	low throup.
#9	CX-6	Bi-	-	-	-	≥3	-	mix of ≤1KB & ≥64KB	-	pause frame
#10	CX-6	Bi-	RC WRITE	-	≥64	-	-	mix of ≤1KB & ≥64KB	≥≈320	pause frame
#11	CX-6	Bidirectional cross-socket traffic on particular AMD servers								pause frame
#12	CX-6	Particular GPU-Direct RDMA traffic on particular servers								pause frame
#13	CX-6	Co-existence of loop traffic and receiving traffic								pause frame
#14	P2100	Bi-	RC	4K	-	≥4	-	-	≥≈1300	low throup.
#15	P2100	-	UD SEND	-	-	-	≥64	-	≥≈32	pause frame
#16	P2100	-	RC READ	1K	≥8	-	-	-	≥≈500	pause frame
#17	P2100	-	RC SEND	-	≤16	-	≥128	≤1KB	≥≈64	pause frame
#18	P2100	Bi-	RC	1K	≥32	-	-	≤64KB	≥≈30	pause frame

Table 2: Performance anomalies found on subsystem F and H with the necessary conditions to trigger them. Anomalies marked with green color are new anomalies found by Collie. Rest are the anomalies we know before building Collie.

latency sensitive, the timeout value can be set high, which allows a larger batch size. Currently the timeout value is set small because most applications supported by this library are latency sensitive. However, by changing this value we successfully enlarge the WQE batch size and the conditions of #10 are all met. This shows the importance of the anomalies found by Collie, as well as how Collie can capture those anomalies missed by existing solutions.

We try our best to reproduce the anomalies found by Collie using existing workload generators (e.g., `Perftest`), only 4 of them (#3, #8, #13, #15) can be reproduced with very careful parameters tuning. Rest anomalies are all outside the search space of existing approaches.

7.2 Running Time for Anomaly Search

To evaluate the efficiency of performance anomaly search, we compare Collie with two baselines: (1) random input generation in our search space and (2) Bayesian Optimization (BO), a widely used method in search problem [31]. We implement the BO approach based on [31]. We set the counter values as BO's optimization target. Our vendors provide us with 9 diagnostic counters. For Collie and BO, we first generate 10 random points. We then compute the standard deviation over the mean of the counter values collected in the first 10 run and use the result to rank these diagnostic counters in decreasing order. Both Collie and BO optimize each diagnostic counter in this order. For a fair comparison, we use MFS to enhance BO as well. In this section, we use subsystem F as an example. We run each search for 10 hours.

Figure 4 shows the running time to find performance anomalies. Random input (i.e., fuzzing) can already find 7 anomalies that only require simple conditions to trigger. BO does improve efficiency but to a very limited extent. BO can speed up the search process but only find 8 anomalies with

the given time. We analyze the optimization process of BO and find that it is not able to optimize the corresponding counters. Our guess is that BO works well when counter values are smooth in the search space. However, the counter values in our search space can have sudden changes, because some discrete dimensions have a huge impact on the counter values (e.g., QP type). Collie uses a simulated annealing based algorithm to optimize the counter values and successfully speed up the search process. Given limited time, it can find all the performance anomalies of this RDMA subsystem. We believe this improvement comes from the optimization process: driving counters to extreme regions is more likely to trigger performance anomalies. It is possible that a more efficient search algorithm (e.g., a fine-tuned BO, reinforcement learning) can perform better, and it is worth future exploration. However, our goal here is to demonstrate that existing simple optimization algorithms, such as simulated annealing, can search efficiently with these hardware counters.

Collie uses diagnostic counters and MFS to further speed up the search. Now we break down their contribution to our overall search speed. Figure 5 shows the result.

The value of diagnostic counters. Figure 5 shows that with performance counters, Collie (Perf) has already found 11 of the 13 anomalies, including the 3 existing ones. This proves that the performance counters are informative and can be used to improve search efficiency. It shows the generality of Collie because performance counters are general and provided by all commodity RDMA subsystems. Figure 5 also shows that using diagnostic counters can further improve the speed. Given limited time, Collie (Diag) can uncover more anomalies and is faster. For example, Anomalies #7 and #8 are not captured by Collie (Perf) because there is no performance change during the search, but Collie (Diag) can observe the
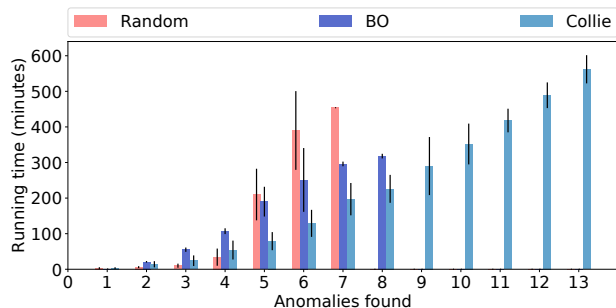
Figure 4: Mean time to find anomalies with random input generation, BO, and Collie. Error bars denote standard deviations. There is no red bar starting from 8, and no purple bar starting from 9, because random input generation and BO can only find 7 and 8 anomalies, respectively.

increase of RNIC internal cache miss and uncover them.

The value of minimal feature set (MFS) The main difference between SA and Collie is whether MFS is applied. With MFS, the efficiency of all approaches (both using diagnostic counters and using performance counters) is significantly improved. For example, Collie (Diag) only uses about half of the time to uncover all the anomalies found by SA(Diag). MFS improves efficiency by eliminating redundant tests from the search space. Otherwise, approaches without MFS may be stuck in the area of an uncovered anomaly.

To understand why increasing diagnostic counter values can help to find anomalies and how MFS works, here we use *Receive WQE Cache Miss* counter as an example. We do not rely on the meaning of these diagnostic counters during the search. To the best of our knowledge, the counter means the number of times that RNICs need to issue extra DMA operations to fetch receive WQE from host DRAM.

Figure 6 shows the diagnostic counter values during the search. The random input generation approach (the orange line) does not increase the diagnostic counter value and thus cannot find many performance anomalies. Collie w/o MFS (the green line) can drive the diagnostic counter value very high, but it cannot find many distinct performance anomalies because further increasing the counter value in the neighboring regions of existing performance anomalies wastes time. Collie (the blue line) is effective in finding performance anomalies, because it can both increase the diagnostic counter value to find application workloads that cause anomalies and also do not need to test application workloads that belong to the same anomaly. Figure 6 shows that most anomalies are found when the diagnostic counter value is high. This also supports the intuition that it is likely to trigger performance anomalies when the diagnostic counter value is driven to extreme regions, which indicates the RDMA subsystem is under pressure. Some anomalies in Figure 6 do not show a high value of this counter. This is mainly due to that they are anomalies that can be easily triggered. They are usually triggered at the beginning of the search process (left corner of Figure 6) and another corresponding diagnostic counter value
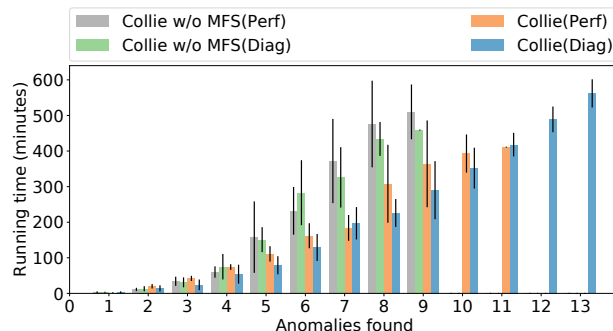
Figure 5: Mean time to find anomalies. (Diag) means diagnostic counters, and (Perf) means performance counters. Error bars denote standard deviations.

is high. For example, Anomaly #13 has simple triggering conditions and is usually found very soon. It does not increase the *Receive WQE Cache Miss* counter but will increase another counter, the counter of *PCIe Internal Back Pressure*.

7.3 Using Collie for Application Design

We use Collie in the development and performance debugging of two key RDMA applications.

First, Collie provides design suggestions for our self-developed efficient RDMA RPC library during its design and implementation. The library needs to be CPU-efficient, and we thus only consider RC as the transport because it is the only transport that supports all one-sided RDMA operations (i.e., READ, WRITE) and ensures reliable messages. In addition, major services that use this RPC library will mainly be deployed on subsystem B and C. Given the search space, Collie provides two suggestions to the developers. (1) Anomaly #4 is in the restricted search space if the RDMA RPC library uses READ, large WQE batch size, and a long SG list to improve throughput and shape the message format. (2) The library needs to use SEND/RECV to deliver small control messages and generally keeps a large receive queue in case of receive-not-ready error. This can potentially trigger Anomaly #5. Unfortunately, both #4 and #5 temporarily have no fix, so Collie suggest developers (1) use RDMA WRITE to transmit data in a batch and (2) configure receive queue depth carefully in SEND/RECV for small control messages transmission. This RDMA based RPC library achieves expected performance and is currently supporting three major services in production.

Second, Collie helps an distributed machine learning (DML) application based on BytePS [14] bypass anomalies during its further development in our production environment. Our DML application encountered anomaly #9 when deploying on our new subsystem E. We worked with multiple vendors (RNIC, server, CPU), but for several weeks we didn't find the root cause or the fix for this anomaly. During this time, we ran Collie and compared the anomalous application with the MFS we got. We found that the application's behaviors matched one of the MFS: (1) use a long SG list to send tensors with several meta data and (2) the message pattern of
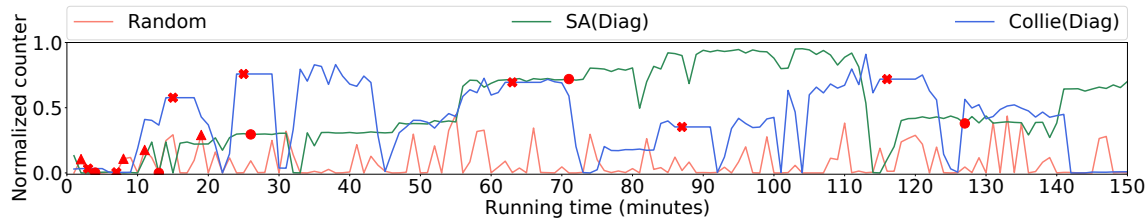
Figure 6: Diagnostic counter values (*Receive WQE Cache Miss*) during the search. Counter values are normalized based on the maximum value we observed in the search. Red crossings denote the performance anomalies found by Collie. Red triangles denote the performance anomalies found by random input generation. Red squares denote the performance anomalies found by Collie without MFS. Collie (the blue line) is flat for a few minutes after finding a new performance anomaly. This is to represent the time needed for extracting the MFS.

tensors and meta data is a typical pattern that contains mix of short and long messages. Collie suggested the developers to avoid these conditions. The developers hence bypassed this anomaly before vendors' fix is ready.

7.4 Implications of the Performance Anomalies Found

After careful analysis of the anomalies found by Collie, we have several interesting and important observations.

Holistic performance testing/tuning over entire RDMA subsystems is important. With our vendors' help, we try our best effort to present the root causes of these anomalies in Appendix A. The root causes can be bottlenecks from RNIC internals, PCIe controllers, and host topologies (cross socket communication). This is because the RDMA network performance is highly related to the entire subsystem and the holistic test is thus important. Besides, we need to configure systems carefully (MTU, PCIe, NUMA, IOMMU, etc.) to fully leverage RDMA's performance [17, 30]. Collie shows that it is sometimes difficult to choose what configuration to use. For example, comparing the Anomaly #14 with other cases related to the MTU setting (e.g., #6), we observe there is no optimal MTU setting for all types of RDMA subsystems. This also indicates that data center operators have to test various RDMA subsystem configurations and tune the system carefully before deploying them.

Opaque resource limitation of the RDMA subsystems. RDMA virtualization, especially performance isolation is important for deploying RDMA to the public cloud environment. Researchers have spent a lot of effort and proposed several solutions [12, 19, 36, 43, 45]. However, anomalies found by Collie suggest that there are new challenges. Existing approaches mainly focus on the isolation of visible resources like *verbs* structures (e.g., QP, MR, CQ), pinned memory, and bandwidth. However, there exist resources that are opaque for developers and data center operators. For example, the RNIC has limited caches that store many data structures, including connection context (well known as QPC) and receive WQE. Anomalies #1, #3, #4, #5 show that severe WQE cache miss can have a huge impact on performance. Hence, it is possible that a connection with a specific message pattern affects another connection by triggering cache misses, even when the bandwidth and other resources are well isolated. We therefore believe it is necessary to take these invisible resources into

consideration when enforcing RDMA performance isolation, especially in public clouds.

Does Ethernet-based RDMA need end-to-end flow control? Currently there is no end-to-end flow control mechanism (e.g., the sliding window for TCP) for production Ethernet-based RDMA deployment (i.e., RoCEv2). Collie shows that this is a major barrier for RDMA subsystems to achieve high-performance and reliability. For example, many anomalies (e.g., #9 and #12) show that the host limitation can slow down RNIC's outbound rate (dispatching received data to host memory). This makes the receiver cannot consume packets as fast as the sender sends. Without end-to-end flow control, the RoCEv2 now can only rely on PFC, the hop-by-hop flow control mechanism. PFC helps to avoid such overflow packet drop but can cause catastrophic consequences [11, 13]. Note that RDMA congestion control [20, 28, 46] mainly targets in-network congestion, so it is orthogonal. A similar observation has been shown in IRN [29], but they mainly focus on in-network behaviors. Collie shows that, in addition to switches, the hosts can also generate PFC pause frames, which requires attention when deploying RDMA in production.

8 Discussion and Future Work

Search space. Collie mainly focuses on how specific application workloads can stress the RDMA subsystems and trigger performance anomalies. We therefore focus on a simple setting of two RNICs and assume the network is free of anomaly. In addition, we temporarily ignore control path behaviors and the inter-arrival time between requests of a connection. The main reason is that adding these factors substantially enlarge the size of our search space. How to efficiently expand Collie's search space is an interesting direction for exploration.

Search algorithm. Collie uses simulated annealing based algorithm with minimal feature set (MFS) to search efficiently. Though powerful data centers can run Collie on multiple machines for a longer time, the search algorithm is also important. According to the MFS found by Collie, the expected time for a random approach is tens of days to find some anomalies that require complicated triggering conditions. There are many other search algorithms alternatives that can be leveraged, such as reinforcement learning. Integrating more search algorithms into Collie is another interesting direction to explore.

Generality of Collie. We believe that Collie can be used for

---

any type of RDMA subsystem or even subsystems with other types of NICs. For example, though the link/transport protocols are different for Infiniband and RoCEv2, the NIC internal structures should be similar (e.g., both can use Mellanox CX-6 VPI RNIC). Collie only relies on non-proprietary counters that expose NIC internal status. Therefore, this methodology should be generalizable to any NIC in any deployment environment if similar counters are available.

Analysis of Performance Anomalies. Collie is designed to uncover anomalies and help to bypass them from the perspective of data center operators, so it assumes minimal hardware knowledge of RDMA subsystems for generality and does not directly analyze the underlying causes. However, since the anomalies found by Collie can be severe (e.g., triggering PFC pause storms), we believe to fully understand them is also an important direction to explore. For example, as mentioned in §7.4, many anomalies are due to bottlenecks on some opaque resources. Both RNIC vendors and data center operators hence need to understand what extra resources should be considered if they want to provide performance isolation for RDMA in a public cloud.

9 Related Work

Hardware bottlenecks in host networking. With the fast growth in NIC performance, researchers have noticed several potential hardware bottlenecks in host networking. Neugebauer et al. [30] study the implication of PCIe performance in host networking. Farshin et al. [6] examine when and when not Intel Data Direct I/O technology can speed up host networking by allowing NIC to access CPU's last-level cache directly. Kalia et al. [15] observe the scalability bottlenecks of caching per-connection metadata in RNIC. Stanko et al. [32] study how the number of connections and memory regions affect performance. These works have raised our attention to RNIC hardware behaviors. Our work is on a different angle: we systematically uncover the performance anomalies that can be triggered by specific application workload due to hardware bottlenecks.

Fuzz testing. Our techniques are in the broader category of fuzz testing. There are three types of fuzz testing: black-box [25, 26], white-box [7, 9, 10], and gray-box fuzzing [1, 40]. Black-box fuzzing is to generate random inputs to test a program, and usually black-box fuzzing can only uncover shallow bugs. In our context, this is also true that using randomly generated application workload can only uncover a small set of anomalies (§7). White-box fuzzing is to use symbolic execution on source code to guide the fuzzer to generate inputs that can have high coverage. We do not have the internal designs of the various components within an RDMA subsystem, so we cannot use white-box approaches. Gray-box fuzzing in the software context is to use the coverage in the control flow graph to guide the fuzzer to incrementally generate inputs that can lead to larger coverage. Our approach is similar to gray-box fuzzing that we both use simulated annealing and mutation-based test case generation. However, the key difference is that we use hardware counters in the RDMA subsystem to guide the search rather than the coverage on the control flow graphs of the source code.

Application design on top of RDMA. Many RDMA application designs leverage specific RDMA performance characteristics, and some already try to circumvent certain RNIC performance anomalies. HERD [16] uses UD SEND and UC Write to implement an RPC library for reduced RNIC packet processing overheads and better scalability. FaSST [18] and eRPC [15] uses UD to further mitigate RNIC scalability bottlenecks in RPC libraries. Kalia et al. [17] provide guidelines to optimize HERD's transport by considering PCIe bottlenecks. FaRM [4, 5] uses RC to access remote in-memory key-value stores, so that it can use RDMA 1-sided READ/WRITE operation for reduced CPU overheads. Our goal is complementary: we systematically uncover the set of performance anomalies of RDMA subsystems that application developers need to be aware of. We show that for RDMA developers, in reality, there is no optimal choice for a particular design decision (e.g., all transport types have certain performance anomalies). Developers therefore need to have a holistic view of all the design decisions and the entire RDMA subsystem before designing and implementing RDMA applications.

10 Conclusion

RDMA has been increasingly used in the industry for its low latency and reduced CPU overheads. Performance anomalies hurt application performance and can lead to catastrophic consequences (e.g., deadlocking the data center network). We build Collie, a tool to help RDMA users to find performance anomalies of the entire RDMA subsystems, without the need for access to any hardware internals design. Collie constructs a comprehensive search space for RDMA application workloads and finds performance anomalies by using simulated annealing to optimize two types of vendor-provided counters. We evaluate Collie on 8 commodity RDMA subsystems and Collie found 15 new performance anomalies that are all acknowledged by the vendor. 7 of them are already fixed under vendors' guidance. We also present our experience in using Collie to guide our development of an RDMA RPC library and help our distributed machine learning applications bypass performance anomalies before vendor fix is ready. Collie is available at `https://github.com/bytedance/Collie`.

References

[1] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed Greybox Fuzzing. In *CCS*, 2017.

[2] Chiranjeeb Buragohain, Knut Magne Risvik, Paul Brett, Miguel Castro, Wonhee Cho, Joshua Cowhig, Nikolas Gloy, Karthik Kalyanaraman, Richendra Khanna, John Pao, et al. A1: A Distributed In-Memory Graph Database. In *SIGMOD*, 2020.

[3] Youmin Chen, Youyou Lu, and Jiwu Shu. Scalable RDMA RPC on Reliable Connection with Efficient Resource Sharing. In *EuroSys*, 2019.

[4] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In *NSDI*, 2014.

[5] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *SOSP*, 2015.

[6] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. Reexamining Direct Cache Access to Optimize I/O Intensive Applications for Multi-hundred-gigabit Networks. In *USENIX ATC*, 2020.

[7] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-Based Directed Whitebox Fuzzing. In *ICSE*, 2009.

[8] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, Fei Feng, Yan Zhuang, Fan Liu, Pan Liu, Xingkui Liu, Zhongjie Wu, Junping Wu, Zheng Cao, Chen Tian, Jinbo Wu, Jiaji Zhu, Haiyong Wang, Dennis Cai, and Jiesheng Wu. When Cloud Storage Meets RDMA. In *NSDI*, 2021.

[9] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-Based Whitebox Fuzzing. In *PLDI*, 2008.

[10] Patrice Godefroid, Michael Y. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. In *NDSS*, 2008.

[11] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over Commodity Ethernet at Scale. In *SIGCOMM*, 2016.

[12] Zhiqiang He, Dongyang Wang, Binzhang Fu, Kun Tan, Bei Hua, Zhi-Li Zhang, and Kai Zheng. MasQ: RDMA for Virtual Private Cloud. In *SIGCOMM*, 2020.

[13] Shuihai Hu, Yibo Zhu, Peng Cheng, Chuanxiong Guo, Kun Tan, Jitendra Padhye, and Kai Chen. Deadlocks in Datacenter Networks: Why Do They Form, and How to Avoid Them. In *HotNets*, 2016.

[14] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters. In *OSDI*, 2020.

[15] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be General and Fast. In *NSDI*, 2019.

[16] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Using RDMA Efficiently for Key-Value Services. In *SIGCOMM*, 2014.

[17] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design Guidelines for High Performance RDMA Systems. In *USENIX ATC*, 2016.

[18] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *OSDI*, 2016.

[19] Daehyeok Kim, Tianlong Yu, Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Shachar Raindel, Chuanxiong Guo, Vyas Sekar, and Srinivasan Seshan. FreeFlow: Software-based Virtual RDMA Networking for Containerized Clouds. In *NSDI*, 2019.

[20] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. HPCC: High Precision Congestion Control. In *SIGCOMM*, 2019.

[21] Ashlie Martinez and Vijay Chidambaram. CrashMonkey: A Framework to Automatically Test File-System Crash Consistency. In *HotStorage*, 2017.

[22] Mellanox. Device Proprietary Counters. https://docs.nvidia.com/networking/display/WINOFv55052000/Device+Proprietary+Counters.

[23] Mellanox. NEO-Host. https://support.mellanox.com/s/productdetails/a2v50000000N2OlAAK/mellanox-neohost.

[24] Mellanox Adapters Programmer's Reference Manual (PRM). https://www.mellanox.com/related-docs/user_manuals/Ethernet_Adapters_Programming_Manual.pdf, 2021.

[25] Barton Miller, Mengxiao Zhang, and Elisa Heymann. The Relevance of Classic Fuzz Testing: Have We Solved This One? *IEEE Transactions on Software Engineering*, page 1–1, 2020.

[26] Barton P. Miller, Louis Fredriksen, and Bryan So. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM*, 33(12):32–44, December 1990.

[27] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *USENIX ATC*, 2013.

[28] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. TIMELY: RTT-Based Congestion Control for the Datacenter. In *SIGCOMM*, 2015.

[29] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting Network Support for RDMA. In *SIGCOMM*, 2018.

[30] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. Understanding PCIe Performance for End Host Networking. In *SIGCOMM*, 2018.

[31] Fernando Nogueira. Bayesian Optimization: Open source constrained global optimization tool for Python. https://github.com/fmfn/BayesianOptimization, 2014.

[32] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pismenny, Liran Liss, Michael Wei, Dan Tsafrir, and Marcos Aguilera. Storm: A Fast Transactional Dataplane for Remote Data Structures. In *SYSTOR*, 2019.

[33] OSU benchmarks. https://mvapich.cse.ohio-state.edu/benchmarks/, 2021.

[34] OFED perftest. https://github.com/linux-rdma/perftest, 2021.

[35] IEEE DCB. 802.1Qbb - Priority-based Flow Control. https://1.ieee802.org/dcb/802-1qbb/, 2021.

[36] Jonas Pfefferle, Patrick Stuedi, Animesh Trivedi, Bernard Metzler, Ionnis Koltsidas, and Thomas R. Gross. A Hybrid I/O Virtualization Framework for RDMA-Capable Network Interfaces. In *VEE*, 2015.

[37] Kun Qian, Wenxue Cheng, Tong Zhang, and Fengyuan Ren. Gentle Flow Control: Avoiding Deadlock in Lossless Networks. In *SIGCOMM*, 2019.

[38] Linux rdma-core. https://github.com/linux-rdma/rdma-core, 2021.

[39] Waleed Reda, Marco Canini, Dejan Kostić, and Simon Peter. RDMA is Turing complete, we just did not know it yet!, 2021.

[40] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *USENIX Security*, 2017.

[41] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. Fast and Concurrent RDF Queries with RDMA-Based Distributed Graph Exploration. In *OSDI*, 2016.

[42] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F. Wenisch, Monica Wong-Chan, Sean Clark, Milo M. K. Martin, Moray McLaren, Prashant Chandra, Rob Cauble, Hassan M. G. Wassel, Behnam Montazeri, Simon L. Sabato, Joel Scherpelz, and Amin Vahdat. 1RMA: Re-Envisioning Remote Memory Access for Multi-Tenant Datacenters. In *SIGCOMM*, 2020.

[43] Shin-Yeh Tsai and Yiying Zhang. LITE Kernel RDMA Support for Datacenter Applications. In *SOSP*, 2017.

[44] Jilong Xue, Youshan Miao, Cheng Chen, Ming Wu, Lintao Zhang, and Lidong Zhou. Fast Distributed Deep Learning over RDMA. In *EuroSys*, 2019.

[45] Yiwen Zhang, Yue Tan, Brent Stephens, and Mosharaf Chowdhury. Justitia: Software Multi-Tenancy in Hardware Kernel-Bypass Networks. In *NSDI*, 2022.

[46] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion Control for Large-Scale RDMA Deployments. In *SIGCOMM*, 2015.

A   Performance Anomalies Found

More details of these anomalies and the lesson we learn are included in this section. We present a concrete example of each anomaly and try our best to simplify each anomaly so that they can be reproduced easier. It is possible to find milder or stricter conditions that trigger the anomaly. We, to the best of our knowledge, also categorize these performance anomalies to their root causes based on our observation and conversations with our vendors.

A.1   Subsystem F with Mellanox 200 Gbps CX-6 VPI

Root cause #1: Receive WQE cache misses bottleneck RNIC receiving rate.

(New) Anomaly #1: UD with large WQE batch size and long WQ causes PFC pause frames and drastic throughput drop. Collie observes that the pause duration ratio can be up to ≈ 20.0% with only a single UD QP. The pause duration ratio means that RNIC is asking the corresponding switch port to pause for ≈ 200 milliseconds within one second on average. We share the NIC vendor with our traffic engine tool and the running command. They have reproduced the anomaly in their environments, but the root cause is still not clear yet. Therefore, we claim this anomaly not fixed yet. To the best of our knowledge, it is likely due to the cache miss triggered by the pre-fetch mechanism for the receive WQE. This bottlenecks the receiver from receiving traffic.

Here is a simplified concrete trigger setting of Anomaly #1: There is 1 connection of UD QP using SEND/RECV Opcode. Each QP has 1 sending MR of 64KB and 1 receiving MR of 64KB. Each QP has a work queue of length 256 (i.e., *max_send/recv_wr* = 256). The MTU is 2KB. The sender keeps sending 64 requests in a batch. Each request only has one SG element and a fixed size of 2KB.

(New) Anomaly #2: UD with small WQE batch size, long WQ, small messages, and a few connections causes throughput to drop without pause frames.

This anomaly is similar to #1 but more tricky and has a different end-to-end symptom. Unlike #1, Collies does not observe PFC pause frames when the setting is slightly different from #1: if the sender does not post sending requests in batch or the batch size is small (e.g., less than 8) and the messages are relatively small (e.g., 512B, 1KB), the throughput will drop by more than 20% without any PFC pause frame triggered when the receiver has an extremely long work queue. If we set a smaller work queue for the receiver, the throughput returns to the line rate. This anomaly is also reproduced and acknowledged by NIC vendor. We conjecture that it has a similar root cause to #1, but due to unknown RNIC bottlenecks, it behaves differently that the throughput drops without pause frame.

Here is a simplified concrete trigger setting of Anomaly #2: There are 16 connections of UD QP using SEND/RECV Opcode. Each QP has 1 sending MR of 64KB and 1 receiving MR of 64KB. Each QP has a work queue of length 1024. The MTU is 1KB. The sender keeps sending 4 requests in a batch. Each request only has one SG element of 1KB.

(New) Anomaly #3: RC READ with large messages causes PFC pause frames when MTU is under 1500 (the default MTU for Ethernet).

We observe the throughput drops drastically once we use RDMA READ opcode with 1500 MTU (1024 for RDMA), the default value for our data centers. The pause duration can be up to 10% and throughput drops to less than half. We report this to our NIC vendor and they tell us the low MTU may trigger the RNIC internal packet processing bottleneck for this 200 Gbps NIC. We carefully survey the potential effect of MTU modification in our deployment and modify the MTU from 1500 to 4200, which supports 4096 as RDMA MTU. This anomaly is successfully fixed in this way.

Here is a simplified concrete trigger setting of Anomaly #3: There are 8 connections of RC QP using Read opcode. Each QP has 1 sending MR of 4MB and 1 receiving MR of 4MB. Each QP has a work queue of length 128. The MTU is 1KB. The sender keeps sending RDMA READ requests. Each request only has one SG element and a fixed size of 4MB.

(New) Anomaly #4: Bidirectional RC READ with large WQE batch size, long SG list, and a few connections causes PFC pause frames, even when MTU is set to 4200 (4096 for RDMA).

This anomaly is tricky but severe. Even with 4200 MTU (Anomaly #3 is solved), Collie observes about 30% PFC pause duration ratio that when bidirectional RDMA READ happens and both sides post a large number of requests in a batch (e.g., 32), each request consists of multiple scatter gather element (e.g., 4) and there are a few connections (e.g., ≈ 160). As usual, this newly found anomaly is reported to the vendor and they have reproduced and confirmed the anomaly. For now, the root cause of this anomaly is still unknown. Therefore, we claim this anomaly not fixed yet.

Here is a simplified concrete trigger setting of Anomaly #4: There are 80 connections of RC QP using Read opcode for each direction. Each QP has 1 sending MR of 64KB and 1 receiving MR of 64KB. Each QP has a work queue of length 128. The MTU is 4KB. The sender keeps sending 128 requests in a batch. Each request has 4 SG elements and a fixed size of 128B.

(New) Anomaly #5: RC SEND with small MTU, large WQE batch, long WQ, and long messages causes PFC pause frames and drastic throughput drop.

(New) Anomaly #6: RC SEND with small MTU, small WQE batch, large SG list batch, long WQ, small messages, and a few connections causes reduced throughput without any pause frame.

They are similar to UD ones (Anomaly #1 and #2) but have a more complex and stricter trigger. For example, Collie observes such anomaly only when MTU is small (e.g., 1024 for RDMA), work depth exceeds 1K for each QP as well as

post multiple receive WQE in a batch. These anomalies are different because they have different QP types and stricter trigger conditions. For example, those anomalous application workloads in #1 and #2 won't trigger anomalies if we only switch the type of QP from UD to RC. Several discussion with our vendors tells us that the *Reliable Connection* type contains some subtle variance inside the RNIC that result in such difference. These two are currently not fixed yet.

Here is a simplified concrete trigger setting of Anomaly #5: There is 1 connection of RC QP using SEND/RECV opcode. Each QP has 1 sending MR of 64KB and 1 receiving MR of 64KB. Each QP has a work queue of length 1024. The MTU is 1KB. The sender keeps sending 64 requests in a batch. Each request has 2 SG elements and a fixed size of 2KB.

Here is a simplified concrete trigger setting of Anomaly #6: There are 32 connections of RC QP using SEND/RECV opcode. Each QP has 1 sending MR of 64KB and 1 receiving MR of 64KB. Each QP has a work queue of length 1024. The MTU is 1KB. The sender keeps sending 8 requests in a batch. Each request has 2 SG elements and a fixed size of 1KB.

Root cause #2: Interconnect Context Memory cache misses reduce RNIC sending rates.

(New) Anomaly #7: RC WRITE with many QPs, small messages, small WQ depth, and small WQE batch size causes reduced throughput.

(New) Anomaly #8: RC WRITE with many MRs, small messages, and small WQE batch size causes reduced throughput.

Though these two anomalies are well-known as the RDMA scalability problem, our real applications do not meet them even when the number of QPs exceeds 10K and the number of MRs exceeds 100K. However, Collie uncovers these two so we classified them into *New* anomalies. We take a deep look into how Collie discovers them and have many discussions with our vendors. We find our experience interesting and worthy of sharing: RNIC caches many necessary structures on its cache (e.g., memory translation table and connection context). When a request triggers cache miss, the RNIC has to issue extra PCIe operation to fetch them from the host DRAM. This will certainly induce extra PCIe latency for processing this request (victim request). However, RNIC is highly pipelined, so even when the victim request has finished the PCIe operation, it may still have to wait for the other pipeline stages to get ready (e.g., a previous long egress request blocks this short egress request). Therefore, if the request size is relatively large enough, the cache miss will not have a large effect on end-to-end performance because the overhead is hidden due to the pipeline.

Here is a simplified concrete trigger setting of Anomaly #7: There are 480 connections of RC QP using RDMA WRITE opcode. Each QP has 1 sending MR of 64KB and 1 receiving MR of 64KB. Each QP has a work queue of length 16. The MTU is 1KB. The sender keeps sending requests without WQE batch. Each request has 1 SG element and a fixed size of 512B.

Here is a simplified concrete trigger setting of Anomaly #8: There are 24 connections of RC QP using RDMA WRITE opcode. Each QP has 1024 sending MR of 64KB and 1024 receiving MR of 64KB. Each QP has a work queue of length 128. The MTU is 1KB. The sender keeps sending requests without WQE batching. Each request has 1 SG element and a fixed size of 512B.

Root cause #3: PCIe controller blocks RNIC from reading host memory.

(Old) Anomaly #9: Bidirectional traffic with a mixture of small and large messages in an SG list on particular AMD servers causes PFC pause frames and drastic throughput drop.

This anomaly is found by one of our production applications that keeps sending such message patterns (described in 2). The root cause of this anomaly is due to PCIe ordering issue. If the RNIC on the AMD server is not configured as PCIe relaxed ordering device, a DMA request may be blocked by the previous one. Therefore, when bidirectional traffic with a mix of short and long requests. The ingress short requests, together with the completion of egress traffic, blocks the ingress long requests. This results in RNIC buffer accumulation and triggers a large amount of PFC pause frames. The throughput can only achieve 60 Gbps with 25% pause frame duration ratio on average. With much effort from our appreciative vendors, we finally fix this by configuring RNIC as a forced relaxed ordering PCIe device.

Here is a simplified concrete trigger setting of Anomaly #9: There are 8 connections of RC QP using RDMA WRITE opcode for each direction. Each QP has 1 sending MR of 4MB and 1 receiving MR of 4MB. Each QP has a work queue of length 128. The MTU is 4KB. The sender keeps sending 8 requests in a batch. Each request has 3 SG elements and the pattern is [128B, 64KB, 1KB].

Root cause #4: RNIC packet processing bottleneck.

(New) Anomaly #10: Bidirectional RC Write with large WQE batch size, a mixture of long messages and lots of short messages, and a few connections causes PFC pause frames.

Collie finds that when several RC QPs keep posting multiple short requests (e.g., 64B, 128B) in batch and a few long requests for both directions, a large amount of pause duration is triggered. This RNIC of the RDMA subsystem has already been configured as forced relaxed ordering PCIe device (Anomaly #8 is solved). Our vendors have confirmed this anomaly and announce it fixed in their upcoming firmware release. The lengthy discussion with our vendor shows us the rough root cause: some component for packet processing inside the RNIC is not fully bidirectional, and our bidirectional reliable traffic (requires packet-level ACK) pattern with a huge amount of short requests, trigger that component's bottleneck. This results in long requests blocked and then many PFC pause frames are generated.

Here is a simplified concrete trigger setting of Anomaly #10: There are 320 connections of RC QP using RDMA

WRITE opcode for each direction. Each QP has 1 sending MR of 64KB and 1 receiving MR of 64KB. Each QP has a work queue of length 128. The MTU is 1KB. The sender keeps sending 64 requests in a batch. Each request has 1 SG element and the pattern is [64KB, 128B, 128B, 128B].

Root cause #5: Host topology causes PCIe latency to increase, and this bottlenecks RNIC receiving rate.

(New) Anomaly #11: On specific types of AMD servers, Bidirectional cross-socket traffic causes pause frame storm and drastic throughput drop.

Collie outputs the minimal feature set with only source/destination NUMA set and bidirectional traffic, indicating these two are the dominant factors. With this bidirectional (A to B and B to A) cross-socket NUMA setting (e.g., NUMA 0 from socket 0 for A and NUMA 2 from socket 1 for B, where socket 0 is the affinitive node for RNIC), even mild traffic with only a single connection can trigger up to 15.7% pause frame duration ratio. After several conversations with our RNIC and server vendors, we conjecture the root cause lies in these particular servers' cross-socket performance because we run the same traffic with the same NIC on different servers but do not observe the same phenomenon. We consider this anomaly as fixed because the vendor helps us roughly understand the root cause and suggest we use 2x100 Gbps NIC (each for a socket) to reduce cross-socket traffic, and we follow this guidance.

Here is a simplified concrete trigger setting of Anomaly #11: There is 1 connection of RC QP using RDMA WRITE opcode for each direction. Each QP has 32 sending MR of 4MB and 32 receiving MR of 4MB. Each QP has a work queue of length 128. The MTU is 4KB. The sender keeps sending 16 requests in a batch. Each request has 1 SG element with a fixed size of 256KB. The QP on host A is using the memory of socket 0 and the QP on host B is using the memory of socket 1.

(Old) Anomaly #12: GPU-direct RDMA causes pause frame storm and drastic throughput drop on particular AMD servers.

We observe a huge amount of pause frames and drastic throughput drop only on some servers in our clusters. The pause duration ratio can be up to 15% and throughput can drop to less than 20% (i.e., 40 Gbps) in this scenario. After careful debugging with our NIC vendor's strong support, we find out that there is a slight difference in PCIe bridge configuration (PCIe ACSCtl) between the anomalous server and normal ones. The anomalous configuration will forward GPU traffic to the root complex rather than directly to the RNIC. We fix this anomaly by adopting the correct configuration.

Here is a simplified concrete trigger setting of Anomaly #12: There are 8 connections of RC QP using RDMA WRITE opcode for each direction. Each QP has 1 sending MR of 4MB and 1 receiving MR of 4MB. Each QP has a work queue of length 128. The MTU is 4KB. The sender keeps sending 8 requests in a batch. Each request has 3 SG elements and the

pattern is [128B, 64KB, 1KB]. All MRs are allocated from GPU memory and we use the GPU under the same PCIe bridge (i.e., shown as PIX/PXB in *nvidia-smi* result).

Root cause #6: RDMA NIC has potential in-NIC incast/congestion.

(Old) Anomaly #13: Co-existence of receiving traffic and loopback traffic causes PFC pause frames.

This anomaly is found in our real applications and can also be uncovered by Collie. Our machine learning system runs workers and servers, and they use RDMA to accelerate the communication. However, once a worker and a server are scheduled on the same physical machine, there will be loopback traffic: the worker will send RDMA traffic to the server on the same host. Meanwhile, the server is receiving traffic from workers on other physical machines. The combination of receiving and loopback traffic triggers congestion/incast inside the NIC. And this RNIC lacks a mechanism to limit the loopback traffic rate, which makes the problem worse. After several discussions with our vendor, we bypass this anomaly by identifying the loopback communication and using other IPC mechanisms (e.g., shared memory). We do not consider this anomaly fixed because we cannot fully rely on other IPC mechanisms, especially for the virtualization environment. This anomaly exposes that a proper design of RNIC needs to consider NIC incast and we are glad to see that some latest RNIC have done so.

Here is a simplified concrete trigger setting of Anomaly #13: There are 16 connections of RC QP using RDMA WRITE opcode. 16 receivers are 8 senders are on the same host A and the other 8 senders are on the host B. Each QP has 32 sending MR of 4MB and 32 receiving MR of 4MB. Each QP has a work queue of length 128. The MTU is 4KB. The sender keeps sending 16 requests in a batch. Each request has 1 SG element with a fixed size of 256KB.

A.2 Subsystem H with Broadcom 100 Gbps P2100G

(New) Anomaly #14: Bidirectional RC traffic with lots of connections and the large MTU causes reduced throughput without PFC pause frame.

Collie observes that a large MTU is necessary to trigger this anomaly. Once we switch the MTU from 4096 (for RDMA) to 1024, both directions can achieve the line rate. This is unusual because most cases show that large MTU improves the performance and small MTU triggers performance anomalies. We don't observe the same phenomenon on any other type of RNICs.

Here is a simplified concrete trigger setting of Anomaly #14: There are 1024 connections of RC QP using RDMA WRITE opcode for each direction. Each QP has 81 sending MR of 256KB and 83 receiving MR of 256KB. Each QP has a work queue of length 128. The MTU is 4KB. The sender keeps sending 1 request in a batch. Each request has 4 SG element with a fixed size of 64KB.

(New) Anomaly #15: UD with long WQ and lots of connec-

tions causes PFC pause frames.

This anomaly is similar to the Mellanox anomaly #1 but has a slightly different trigger. Collie successfully trigger #1 with only a single connection, but for P2100 RNIC our multiple runs show that a few connections are necessary.

Here is a simplified concrete trigger setting of Anomaly #15: There are 32 connections of UD QP using SEND/RECV opcode. Each QP has 1 sending MR of 4KB and 1 receiving MR of 4KB. Each QP has a work queue of length 64. The MTU is 2KB. The sender keeps sending 1 request in a batch. Each request has 1 SG element. The message pattern is like [256B, 1KB, 64B, 1KB].

(New) Anomaly #16: RC READ with lots of connections, large WQE batch size, and small MTU causes PFC pause frames.

This anomaly is similar to the Mellanox anomaly #4 and it shows that for the same RNIC and other hardware components, the best MTU choice can be different when workloads change.

Here is a simplified concrete trigger setting of Anomaly #16: There are 500 connections of RC QP using RDMA READ opcode. Each QP has 1 sending MR of 256KB and 1 receiving MR of 256KB. Each QP has a work queue of length 128. The MTU is 1KB. The sender keeps sending 8 requests in a batch. Each request has 1 SG element with a fixed size of 64KB.

(New) Anomaly #17: RC SEND with lots of connections, small WQE batch size, small MTU, short messages, and long WQ causes PFC pause frames.

We have reported this anomaly to our vendor. To the best of our knowledge, we conjecture this anomaly is related to some corresponding WQE cache component inside RNIC.

Here is a simplified concrete trigger setting of Anomaly #17: There are 80 connections of RC QP using SEND/RECV opcode. Each QP has 1 sending MR of 1MB and 1 receiving MR of 1MB. Each QP has a work queue of length 128. The MTU is 1KB. The sender keeps sending 1 request per batch. Each request has 1 SG element of fixed size 1KB.

(New) Anomaly #18: Bidirectional RC WRITE with a few connections, large WQE batch, and small messages causes PFC pause frames.

Our vendor has confirmed anomalies #17 and #18. They have reproduced these two anomalies and help us fix them. The solution is to configure some specific registers of the RNIC, and these two anomalies disappear.

Here is a simplified concrete trigger setting of Anomaly #18: There are 16 connections of RC QP using RDMA WRITE for each direction. Each QP has 1 sending MR of 12KB and 1 receiving MR of 12KB. Each QP has a work queue of length 64. The MTU is 1KB. The sender keeps sending 16 requests in a batch. Each request has 1 SG element of fixed size 64KB.

SCALE: Automatically Finding RFC Compliance Bugs in DNS Nameservers

Siva Kesava Reddy Kakarla[1]   Ryan Beckett[2]   Todd Millstein[1,3]   George Varghese[1]

[1]*University of California, Los Angeles*   [2]*Microsoft*   [3]*Intentionet*

Abstract

The Domain Name System (DNS) has intricate features that interact in subtle ways. Bugs in DNS implementations can lead to incorrect or implementation-dependent behavior, security vulnerabilities, and more. We introduce the first approach for finding RFC compliance errors in DNS nameserver implementations, via automatic test generation. Our SCALE (Small-scope Constraint-driven Automated Logical Execution) approach *jointly* generates zone files and corresponding queries to cover RFC behaviors specified by an executable model of DNS resolution. We have built a tool called FERRET based on this approach and applied it to test 8 open-source DNS implementations, including popular implementations such as BIND, POWERDNS, KNOT, and NSD. FERRET generated over 13.5K test cases, of which 62% resulted in some difference among implementations. We identified and reported 30 new unique bugs from these failed test cases, including at least one bug in every implementation, of which 20 have already been fixed. Many of these bugs existed in even the most popular DNS implementations, including a critical vulnerability in BIND that attackers could easily exploit to crash DNS resolvers and nameservers remotely.

1 Introduction

The Domain Name System (DNS) plays a central role in today's Internet, as it allows users to connect to online services through user-friendly domain names in place of machine-friendly IP addresses. Organizations across the Internet run DNS *nameservers*, which use DNS configurations called zone files to determine how to handle each query, either returning an IP address, rewriting the query to another one, or delegating the responsibility to another nameserver. There are many popular nameserver implementations of the DNS protocol in the wild, both open-source [21, 23, 25, 76] and in public or private clouds [2, 39, 85, 97].

Over time DNS has evolved into a complex and intricate protocol, spread across numerous RFCs [41, 80, 86, 96]. It is difficult to write an efficient, high-throughput, multithreaded implementation that is also bug-free and compliant with these RFC specifications. As a result, nameserver implementations frequently suffer from incorrect or implementation-specific behavior that causes outages [34, 103, 106], security vulnerabilities [74, 94], and more [15, 19, 22].

This paper presents the first approach for identifying RFC compliance errors in DNS nameserver implementations, by automatically generating test cases that cover a wide range of RFC behaviors. The key technical challenge is the fact that a DNS test case consists of both a query and a zone file, which is a collection of *resource records* that specify how queries should be handled. Zone files are highly structured objects with various syntactic and semantic well-formedness requirements, and the query must be related to the zone file for the test even to reach the core query resolution logic.

Existing standard automated test generation approaches are not suitable for our needs, as illustrated in the top of Figure 1. Fuzz testing is scalable but has well-known challenges in navigating complex semantic requirements and dependencies [13, 36], which are necessary to generate behavioral tests for DNS. As a result, fuzzers for DNS only generate queries and hence are used only to find parsing errors [10, 32, 89, 99]. Symbolic execution [72] can, in principle, generate DNS tests that achieve high code coverage but, in practice, suffers from the well-known problem of "path explosion" [9, 13, 36] that limits scalability and coverage. As a result, symbolic execution has only been used to identify generic errors like memory leaks in individual functions within nameserver implementations, again avoiding the need to generate zone files [93].

Our approach to automated testing for DNS nameservers, which we call **SCALE** (Small-scope Constraint-driven Automated Logical Execution), *jointly* generates zone files and the corresponding queries, does so in a way that is targeted toward covering many different RFC behaviors, and is applicable to black-box DNS nameserver implementations. The key insight underlying SCALE is that we can use the existing RFCs to define a model of the logical behaviors of the DNS resolution process and then use this model to guide test generation. Specifically, we have created an *executable* version of a recent formal semantics of DNS [71], which we then symbolically execute to generate tests for black-box DNS nameservers — each test consisting of a well-formed zone file and a query that together cause execution to explore a particular RFC behavior. Intuitively, tests that cover a wide variety of behaviors in our executable model will also cover a wide variety of behaviors in DNS nameservers since they have the same goal, namely to implement the RFCs.

Symbolic execution of our logical model is still fundamentally unscalable — there are an unbounded number of possible

(a) Fuzz testing

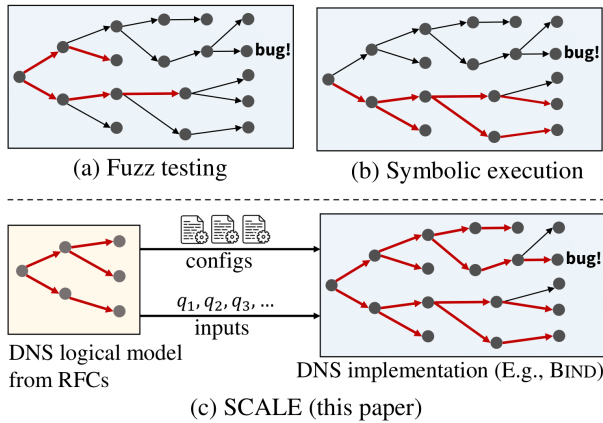(b) Symbolic execution

(c) SCALE (this paper)

Figure 1: Overview of different automated testing approaches. Tested implementation paths are shown in red. (a) Fuzz testing is scalable but is often unable to navigate complex input requirements. (b) Symbolic execution can solve for input conditions but suffers from path explosion and has difficulty with complex data structures and program logic, and will thus only typically explore a small subset of possible program paths. (c) SCALE uses a logical model of the DNS RFCs to guide symbolic search toward *many* different *logical behaviors*.

execution paths, they grow exponentially in the size of the zone file, and expensive constraint solvers must be used to generate a test case for each path. We therefore bound the generated zone files to contain a very small number of resource records and short domain names — a maximum of 4 for each of these in our experiments, which is much smaller than real-world zone files. However, we provide experimental evidence of the existence of a *small-scope* property [43], meaning that many interesting behaviors can be covered with small tests. First, each return point in our logical model can be reached with a test where the length of domain names and the number of records in the zone file is at most 3. Each return point represents a distinct RFC-specified scenario for DNS resolution (e.g., a particular flavor of query rewrite). Second, while increasing this constant from 2 through 4 increased the number of errors that our tool identified, no new errors were found in a sample of paths that required size 5. This finding makes sense because, while zone files can contain a large number of records, the number of records that are relevant to any particular query tends to be small.

We have used the SCALE approach as the basis for a tool called FERRET[1] for automated testing of DNS nameserver implementations (Figure 2). FERRET generates tests using our logical model, which we have implemented in a modeling language called Zen [4] that has built-in support for symbolic execution. FERRET then performs *differential* testing by running these tests on multiple DNS nameserver implementations and comparing their results to one another. In this way FERRET can identify RFC violations, crashes, as well as situations where the RFCs may be ambiguous or underspecified, leading
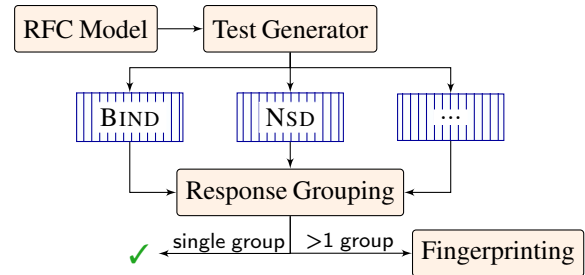
Figure 2: FERRET system architecture.

to implementation-dependent behavior. Because DNS implementers strive for behavioral consistency among their implementations [92], any test that produces divergent results among the implementations represents a likely error. However, there can be orders-of-magnitude fewer root causes than divergent tests, so as a final step we provide a simple but effective technique to help users with bug deduplication. We create a *hybrid fingerprint* for each test, which combines information from the test's path in the Zen model with the results of differential testing, and then group tests by fingerprint for user inspection.

Using FERRET, in just a few hours we generated over 12.5K valid test cases[2] with a maximum zone-file size of 4 records. Running these tests on 8 different open-source DNS nameserver implementations, we found that the implementations' behaviors only completely agreed on 35% of the tests. Our fingerprinting technique reduced the remaining cases to roughly 75 groups. Because our executable model includes a specification of the well-formedness conditions for zone files, we also leveraged Zen to systematically generate zone files that violate one of these conditions. We generated 900 invalid zone files of which 184 resulted in some difference among implementations. Inspecting tests from each fingerprinted group resulted in the discovery of 30 unique bugs across the different implementations. Developers have confirmed all of them as actual bugs and fixed 20 of them, at the time of writing. The most severe bug FERRET found was a subtle combination of zone file and query that an attacker could easily use to crash both BIND nameservers *and* resolvers remotely. We engaged in a secure disclosure process, after which the developers fixed the issue and then publicly disclosed the vulnerability, through a CVE (CVE-2021-25215) [26, 38] rated with high-severity.

Contributions: This paper's contributions are:
- The first automated approach to identify RFC violations in black-box DNS nameservers. A unique feature of our approach, SCALE, is the joint generation of zone files and queries to produce high-coverage behavioral tests.
- An implementation of our approach in FERRET that combines SCALE with differential testing.
- A novel fingerprinting approach for bug deduplication that takes advantage of our RFC model to help triage bugs.
- An evaluation from testing 8 different open-source DNS nameserver implementations with tests generated by FER-
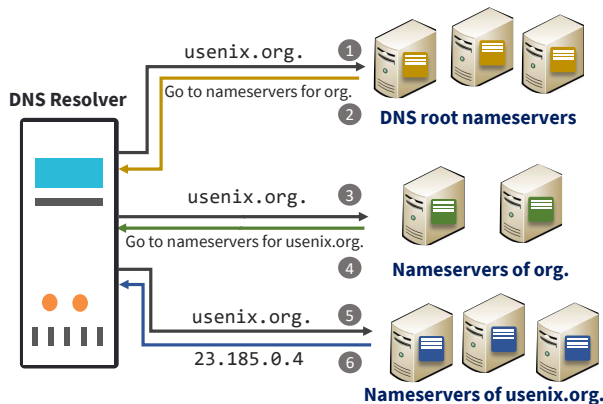
Figure 3: The resolution process for the domain name usenix.org (with no caching).

RET consisting of over 13.5K zone files, which resulted in the discovery of 30 new unique bugs and no false positives.

2 Background And Motivation

In this section, we first give a brief overview of DNS and then motivate FERRET through two previously unknown errors that it found in the popular BIND software for DNS [23].

2.1 Overview of DNS

The Domain Name System (DNS) is the phone book of the Internet. Its primary role is to translate domain names (like usenix.org) into various pieces of information, IP addresses being the most common. A domain name is represented as a sequence of labels joined by the . character. These labels form a tree-like hierarchy with the root as . and org as a child of it and so on. Each label at any level in the hierarchy can contain information, and the user obtains that information by querying the domain name formed by joining the labels from that node to the root. Data is stored as DNS *resource records* where each record has a domain (owner) name, a type for its information, and the content, among other things.

The namespace database tree is divided into a large number of *zones*. A zone is a collection of records that share a common end domain name. For example, the usenix.org zone has only records ending with usenix.org. All the resource records of a zone are available to the user through a set of authoritative nameservers, which are in turn identified by a domain name. For example, the usenix.org zone is available from servers like dns1.easydns.com, dns2.easydns.net and dns3.easydns.ca. The same zone is served by multiple servers to ensure redundancy and availability.

To resolve a domain name like usenix.org to its IP address, a client will traverse the tree from one of the root nameservers. The root nameserver checks its local zone file and either provides the IP record or returns a set of authoritative nameservers to ask instead. The client continues

Example Record			Description
a.exm.org. A	1.2.3.1		IPv4 record
*.exm.org. AAAA	1:db8::2:1		Wildcard IPv6 record
s.exm.org. NS	ns.dns.com.		Delegation record
c.exm.org. DNAME	cs.org.		Domain redirection
w.exm.org. CNAME	a.exm.org.		Canonical name

Table 1: Examples of common DNS record types.

by querying the new set of nameservers either until the query is resolved or gets a non-existent domain name error. The process or the software that performs this traversal on the client side is called a *resolver*. The resolution process for the domain usenix.org is shown in Figure 3.

A nameserver can serve multiple zones. When a query comes to the nameserver, it first checks whether the query ends with any of the zone domains; otherwise, it sends a refusal message to the resolver. After picking a zone, the nameserver will look up the query name's closest matching records. It then creates a response based on the query type and the records selected. DNS supports many record types, including records for IP addresses, pointers to other records, domain aliases, delegation records, and more. Table 1 shows a few example records.

2.2 Finding DNS Errors with FERRET

The goal of FERRET is to automatically generate high-coverage query and zone file inputs to find behavioral errors in DNS nameserver implementations. In this subsection we illustrate both the challenges in doing so and FERRET's capabilities through two example errors that it automatically found in BIND.

Bug #1: BIND sibling glue records bug. FERRET generated the following test case, which identified a previously unknown performance bug in BIND [47].[3]

```
    campus.edu.    SOA    ...
foo.campus.edu.    NS     ns1.campus.edu.
ns1.campus.edu.    A      1.1.1.1
```
Query: ⟨anything.foo.campus.edu., A⟩

In this test case, the query matches the NS record in the zone file, which delegates the query to another nameserver, ns1.campus.edu. However, that nameserver happens to be a sibling of foo.campus.edu (as they are both directly under campus.edu), and the zone file contains an A record, called a *glue record* [41], for the nameserver's IP address. NSD, KNOT, and POWERDNS correctly return the NS record along with the glue record, avoiding extra round-trips to determine the nameserver's IP address, while BIND returns only the NS record. Returning the sibling glue record is not compulsory, but our test case exposed two unrelated errors that can negatively affect the performance of many queries.

---

[3]Note that we have renamed the labels for all the example bugs for clarity.

After we filed the issue the BIND developers confirmed the bug saying, "This report turns out to be very interesting..." Briefly, BIND uses a "glue cache" that had two bugs. First, if the cache lookup fails, then glue records are supposed to be searched for in the zone file, but this was not happening. Second, glue records for siblings domain nameservers were accidentally never searched for at all.

This example illustrates the challenges of identifying nameserver behavior errors. Even though the zone file has only a few records, they have complex dependencies. First, there must be a delegation of the query to another nameserver. Second, that nameserver must be in the same zone. Third, that nameserver must be a sibling domain. Fourth, there must be a glue record for that domain in the zone. Given these dependencies, it is understandable that prior testing techniques did not uncover these bugs. Further, by comparing the outputs from multiple implementations, FERRET is able to identify this test case as potentially buggy behavior despite receiving a valid response from BIND.

Bug #2: BIND crash. As another, more dire example, consider the following zone file that FERRET generated. The zone file is invalid due to having two identical records, but BIND, NSD, and KNOT accept the zone file and make it valid by ignoring the duplicate record.

```
      attack.com.    SOA     ...
      attack.com.    NS      ns1.outside.com.
      attack.com.    NS      ns1.outside.com.
host.attack.com.    DNAME   com.
```
Query: ⟨`host.attack.host.attack.com.`, `DNAME`⟩

FERRET generated multiple queries for this zone file (§ 3.6) and the one showed above caused BIND to crash.

In this test case, the `DNAME` record is applied to rewrite any queries ending with `host.attack.com` to end with just `com`, so the query that FERRET generated is rewritten to the new query `host.attack.com`. The nameservers add the `DNAME` record and rewritten query to the response before resolving the new query. The new query exactly matches the same `DNAME` record, so implementations are expected to return the current response. All implementations except BIND behaved as expected. BIND did not respond, and the query timed out. Inspecting the logs, we found that the server crashed with an assertion failure due to an attempt to add the same `DNAME` record to the response twice.

This error constitutes a critical security vulnerability. We next describe two scenarios to show how this failed assertion check can be exploited remotely by an attacker.

Scenario 1 - **Attack on a DNS hosting service that uses Bind:** DNS hosting services using BIND's authoritative nameserver implementation (e.g., Dyn [42]) are vulnerable to this attack. An attacker can upload the above zone file to the authoritative server instances through the hosting service. Then, when the above query is requested, the server instances will crash as shown in Figure 4(a). Since a server instance

(a) Attack on a DNS hosting service using Bind

(b) Attack on a public Bind DNS Resolver
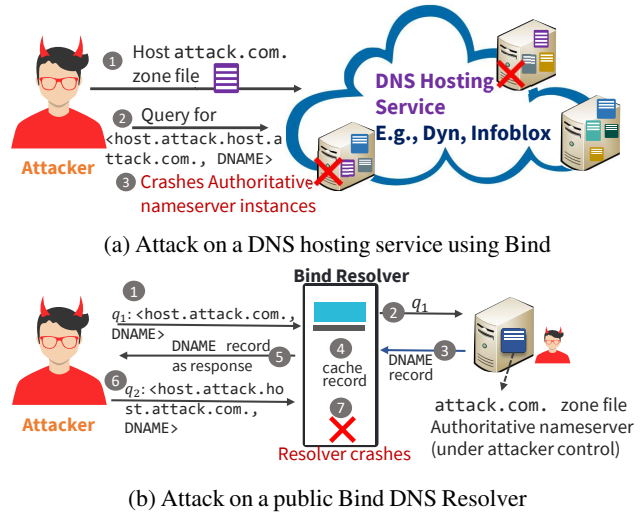
Figure 4: `DNAME` attack targeting the DNS hosting services (a) and the public BIND based recursive resolvers (b).

will generally be serving zone files from multiple customers, such a crash will take down the zones for all customers hosted at that nameserver. This provides a method for attackers to trivially and remotely initiate a denial of service attack against customers hosted by such a service.

Scenario 2 - **Attack on a public Bind DNS resolver:** In this second scenario, the attacker can crash any public DNS resolver based on BIND, thereby constituting, as stated by the BIND security team, an "easily-weaponized denial-of-service vector." As illustrated in Figure 4(b), the attacker purchases, registers, and controls the `attack.com` zone and its authoritative servers. The attacker then simply requests the `DNAME` record from a public recursive resolver running BIND, which attempts to fetch the result from the attacker's authoritative server. This record is cached, and then the test query is sent to the resolver. The resolver uses the cached `DNAME` record and ultimately crashes as described earlier. In some estimates, BIND accounts for over half of all DNS resolvers in use [75], which means that attackers could effectively initiate a simple distributed denial of service (DDoS) attack against the numerous ISPs and public resolvers available to end users.

Disclosure: After discovering the `DNAME` attack, we initiated a responsible disclosure procedure with the BIND maintainers. Understanding the attack severity, they requested that we keep the issue confidential until they worked through their process to patch and then disclose the bug to the relevant parties in a controlled manner. BIND released a Common Vulnerabilities and Exposure (CVE-2021-25215) [26, 38], with a "high severity" rating and asked developers and users to upgrade to the patched version. The attack affected all maintained BIND versions, which in turn affected RHEL, Slackware, Ubuntu, and Infoblox.
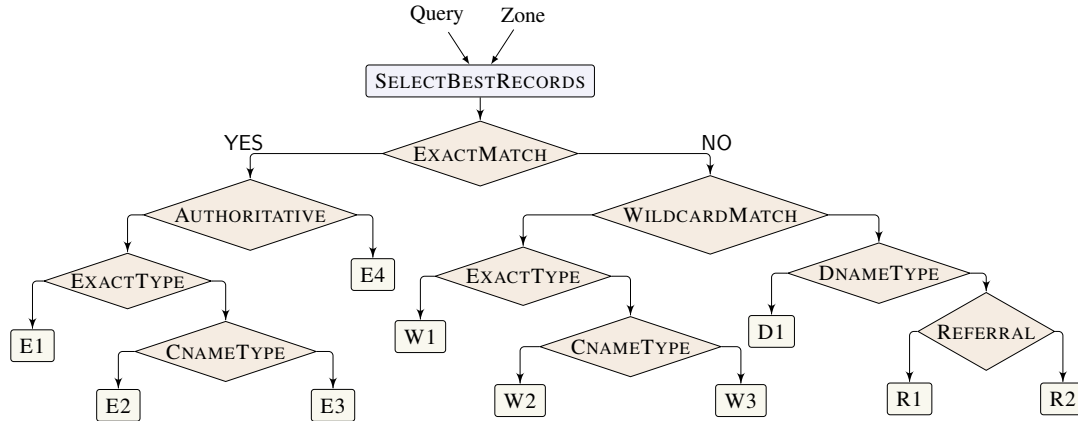
Figure 5: Abstract representation of the Authoritative DNS decision tree used to respond to a user query.

3 Methodology

In this section we overview our methodology for generating high-coverage tests for DNS nameserver implementations and discuss how we address several technical challenges.

3.1 SCALE Approach

As illustrated by the examples in the previous section, the inputs to a DNS nameserver — a query and a zone file containing a set of records — are highly structured. Further, records can be of many different types and have many different kinds of dependencies among them. Therefore, an effective approach to automatically identifying RFC violations must be able to generate *valid* inputs that meet the required structural and semantic constraints of the domain, and it must also be able to explore different combinations of record types and features in a systematic way. To solve this joint generation problem, our approach, SCALE (Small-scope Constraint-driven Automated Logical Execution) leverages a specification of the DNS nameserver logic to drive test generation. Specifically, we have created an *executable* version of an existing DNS specification [71] and generate tests through *symbolic execution* [72] on this executable specification. Symbolic execution is a static analysis technique that enumerates execution paths in a program and uses automated constraint solvers to produce an input that will take each enumerated path, thereby generating tests that cover many different program behaviors.

While the end-to-end behavior of a DNS query lookup can require contacting many nameservers, we employ a *compositional* approach that only generates tests for a single nameserver in isolation. Because our formal model considers the space of all inputs to the nameserver that could be produced by the rest of the system, and because the "next step" delegation of the resolution process is captured in the output at a single nameserver, this approach still allows us to generate tests for all behaviors of the end-to-end DNS. In other words, any implementation bug that exists in a DNS nameserver

implementation can be found using our approach. In general, a downside of compositional testing is that it can lead to false positives if the tester considers input states that are, in reality, unreachable with respect to the rest of the system. However, in the case of DNS, nameservers keep no internal state — the response they provide is based only on the supplied query and configuration. This stateless nature implies that compositional testing will not incur any false positives.

Hence our formal semantics focuses on query lookup at a single nameserver, which we model as a stateless function that takes a user query and a zone file and produces a DNS response. Figure 5 shows an abstract view of this function. Given the input query and zone, DNS will first select the closest matching records in the zone for the query using the SELECTBESTRECORDS function and then follow the decision logic laid out in the figure using these records. Each leaf node represents a unique case in the DNS. For example, the tree shows four different cases of exact matches, labelled E1 through E4. Symbolic execution of our query-lookup function generates inputs that drive the function down different execution paths, thereby enabling us to systematically explore the space of DNS behaviors and feature interactions.

Example: Consider the path in Figure 5 to the leaf labelled R1. In order to reach that leaf, the selected records must not contain one with either an exact match or a wildcard match on the query domain name. Further, there should not be a DNAME match but should be one of type NS (REFERRAL). Finally, while not shown in the figure, when preparing a response to the query the function will also search for a glue record if the NS target is in the same zone. Solving all of these constraints caused symbolic execution to automatically generate the first test case shown in § 2.2, which identified two errors in BIND.

3.2 An Executable Model of DNS

We have created an implementation of the formal semantics of query lookup [71] as a program in a modeling language called Zen [4], a domain-specific language (DSL) embedded in C#.

```
1   Zen<Response> QueryLookup(
2     Zen<Query> q,
3     Zen<Zone> z)
4   {
5     var records = SelectBestRecords(q, z);
6     var rname = records.At(0).Value().Name();
7     var types = records.Select(r => r.Type());
8
9     return If(
10      rname == q.Name(),
11      ExactMatch(records, q, z),
12      If(
13        IsWildcardMatch(q.Name(), rname),
14        WildcardMatch(records, q, z),
15        If(
16          types.Any(t => t == RType.DNAME),
17          Rewrite(records, q),
18          If(
19            And(types.Any(t => t == RType.NS),
20                Not(types.Any(t => t == RType.SOA))),
21            Response(Tag.R1,
22              Delegation(records, z), Null<Query>()),
23            Response(Tag.R2, empty, Null<Query>())
24      )))));
25  }
```

Figure 6: Record lookup model in C# using Zen.

To illustrate this approach, we show several components of our model. Figure 6 shows the model's main query-lookup function, as depicted in Figure 5. The function first selects the best records (Line 5) and then tests if the query domain name is equal to the records' domain name (Line 10). If so, then this is an exact match and the model calls out to a helper function to specifically handle the ExactMatch subcase (Line 11). Similarly, if the query domain name is a wildcard match for the record domain name (Line 13), then we invoke the WildcardMatch subcase (Line 14). We show the implementation of wildcard matching in Figure 7. This function implements the case where the best matching record is a wildcard, properly handles interactions with CNAME records, and synthesizes the correct records for use in the resolver cache.

Our complete executable model consists of 520 lines of C# code. The model can also easily extend to new DNS RFCs that would be added in the future. Similarly, if an organization has a particular way of resolving RFC ambiguities or purposely deviates from the RFCs in specific ways, the organization can modify the logical model to reflect that intent.

We chose to implement our formal model in Zen because it has built-in support for symbolic execution. In Zen, certain inputs can be marked as *symbolic*, and the tool will then leverage SMT solvers [27] to produce concrete values for these inputs that drive the program down different execution paths. In our code examples, the Zen<T> type for inputs has the effect of marking them as symbolic. The tests produced by symbolic execution can then be used to test any DNS nameserver implementation. However, making symbolic execution effective required us to address several challenges, which we describe in the rest of this section.

```
26  Zen<Response> WildcardMatch(
27    Zen<IList<ResourceRecord>> rrs,
28    Zen<Query> q,
29    Zen<Zone> z)
30  {
31    var exact = rrs.Where(r => r.Type() == q.Type());
32    var record = rrs.At(0).Value();
33    var newQuery = Query(record.RData(), q.Type());
34    var exactSyn = RecordSynthesis(exact, q.Name());
35    var cnameSyn = RecordSynthesis(rrs, q.Name());
36
37    return If(
38      exact.Length() > 0,
39      Response(Tag.W1, exactSyn, Null<Query>()),
40      If(
41        rrs.Any(r => r.Type() == RType.CNAME),
42        Response(Tag.W2, cnameSyn, Some(newQuery)),
43        Response(Tag.W3, empty, Null<Query>())
44    ));
45  }
```

Figure 7: Wildcard match model in C# using Zen.

3.3 Generating Valid Zone Files

The first challenge that we encountered is that zone files must satisfy several constraints in order to be considered well-formed. For instance, if there is a DNAME record in a zone file for math.uni.edu, then no other records below this domain name may exist, for any record type (e.g., an A record for fun.math.uni.edu is not allowed). The DNS RFCs define many such constraints as a way to eliminate ambiguous or useless zones, as shown in Table 2. Naively performing symbolic execution will produce many zone files that are not well formed. Further, DNS implementations typically preprocess zone files to reject ill-formed zones, thereby failing to test the intended execution path of the query lookup logic.

Fortunately, our SCALE approach admits a natural solution to this problem. We have formalized all of the DNS zone validity conditions as predicates in Zen. Whenever Zen's symbolic execution engine produces a constraint representing the conditions under which the query lookup function takes a particular execution path, we conjoin these predicates to that constraint before Zen passes it off to an automated constraint solver. In this way we ensure that all test cases will have well-formed zone files by construction.

3.4 Data Representation

In our Zen model, we represent zone files as a list of resource records, where each resource record contains a domain name, record type, and data fields. We represent user queries similarly as consisting of a domain name and a query type. Record and query types are represented using enums, which Zen translates to integer values.

One challenging decision we ran into was how best to represent and model domain names, for both zone records and record data, in a manner that permits fully automatic and scalable analysis. For instance, a natural way to encode domain names

Validity Condition	RFC Document
i. All records should be unique (there should be no duplicates).	2181 [28]
ii. A zone file should contain exactly one `SOA` record.	1035 [87]
iii. The zone domain should be prefix to all the resource records domain name.	1034 [86]
iv. If there is a `CNAME` type then no other type can exist and only one `CNAME` can exist for a domain name.	1034 [86]
v. There can be only one `DNAME` record for a domain name.	6672 [96]
vi. A domain name cannot have both `DNAME` and `NS` records unless there is an `SOA` record as well.	6672 [96]
vii. No `DNAME` record domain name can be a prefix of another record's domain name.	6672 [96]
viii. No `NS` record can have a non-`SOA` domain name that is a prefix of another `NS` record.	1034 [86]
ix. Glue records must exist for all `NS` records in a zone.	1035 [87]

Table 2: Summary of DNS zone file validity conditions specified in various RFCs.

would be as string values (a domain name is just a '.' separated string). Indeed, modern SMT solvers like Z3 [27] support the logical theory of strings, so this is a natural approach to consider. However, the theory of strings is in general undecidable [14, 35]. Moreover, this encoding would require us to define complex predicates for manipulating domain names, including extracting each of the labels of a domain name and checking whether one domain name is a prefix of another.

Therefore, rather than model domain names as strings, we take advantage of the observation that the particular character values in a domain name label string do not matter for DNS lookup. Instead, all that matters is whether two labels are equivalent to one another and whether a label represents a wildcard. As such, we encode a domain name in Zen as a list of integers and use a specific integer value to represent the wildcard character '*'. This allows us to use simple, efficient integer operations and constraints to manipulate domain names according to our formal model.

3.5 Handling Unbounded Data

A final challenge associated with symbolic execution for our formal model is the fact that there are several sources of *unboundedness*. For example, a zone file can contain an unbounded number of records, and a domain name can contain an unbounded number of labels. Our Zen model contains an unbounded number of paths, since the number of resource records in a zone file is unbounded and the function to select the best records must examine all of them and compare them to one another. SMT constraint solvers have limited support for unbounded data structures such as lists, and in general, reasoning about such constraints requires quantifiers, which lead to undecidability [95]. Therefore, in our Zen implementation we only consider inputs that have a bounded size, e.g., at most N records in a zone file, and hence only produce test cases that respect these bounds. The size of inputs is a parameter that is configurable by the user. While the SCALE approach can therefore fail to detect some errors, we provide experimental evidence of the existence of a *small-scope* property [43], meaning that many interesting behaviors, and behavioral errors, can be exercised with small tests (§ 5.1).

3.6 Generating Tests for Invalid Zone Files

While it's critical to be able to generate well-formed zone files for testing, bugs can also lurk in implementations' handling of ill-formed zones. Many DNS implementations use zone-file preprocessors to perform syntactic and semantic checks. For example, BIND uses `named-checkzone` [24], KNOT uses `kzonecheck` [18], and POWERDNS uses `pdnsutil` [20]. The implementations either reject an ill-formed zone or accept it but convert it to a valid one by ignoring certain records that cause it to be semantically ill-formed.

Many security vulnerabilities for software lie in the incorrect handling of unexpected inputs (e.g., in parsers [1]), and DNS software should be no different. Since our executable model includes a formulation of the validity conditions for zone files, we leverage Zen to systematically generate zone files that violate one of these conditions. For example, we ask Zen to generate a zone file in which all but the 7th condition in Table 2 is violated and the rest are satisfied.

If an invalid zone is rejected, then there is no issue, but if it is accepted, then there can be errors in how the zone is used for DNS lookups. To test for such errors we must also be able to generate queries for these zones. However, our formal model is only well defined for valid zone files so we cannot use it to generate queries. Instead, we use a technique from our prior work on zone-file verification [71] to partition queries into equivalence classes (ECs) relative to a given zone file. An equivalence class is a set of queries with the same resolution behavior, assuming a correct underlying DNS implementation, and the ECs are generated through a simple syntactic pass over a zone file. FERRET generates these ECs and then uses one representative query from each EC as a test. Though the number of ECs can vary widely, depending on the records in a zone file, in practice a zone containing four records will typically induce tens of ECs.

4 System Overview

FERRET is divided into several components, which are depicted in Figure 2. First it uses our Zen model described above to generate test inputs. Because domain names are encoded in

Zen using lists of integer labels (see § 3.4), FERRET includes a shim layer that translates the generated zone files and queries into meaningful domain names by mapping these labels to a collection of predefined strings (e.g., com). FERRET uses the equivalence-class (EC) generation algorithm of GROOT [71] to generate test queries for invalid zone files (§ 3.6).

FERRET uses Docker [83] to construct a working container image of each implementation. We cloned the implementations' code as of October 1st, 2020 [16, 21, 23, 25, 29, 33, 76, 102], from their open-source repositories on GitHub [84] and GitLab [100]. FERRET starts a container for each image, and each container serves one zone file at a time as an authoritative zone. FERRET uses a Python library dnspython [17] to construct queries and send them to each implementation's container. For each test case, the Python script prepares the container by stopping the running DNS nameserver, copying the new zone file and the necessary implementation-dependent configuration files to the container, and then restarting the DNS nameserver.

Finally, FERRET performs response grouping followed by fingerprinting to deduplicate errors that are likely to have the same root cause. For each test case, two DNS responses are considered equivalent, and hence in the same group, if they have the same response flags, return code, answer, and additional sections. FERRET only compares the authority section in two responses when their answer sections are empty. We do this because implementations are free to add additional records like a zone's SOA or NS records along with the requested records. We then fingerprint tests that result in more than one group and thereby represent a likely error. The fingerprint for a valid test is a tuple consisting of (1) the case in the formal model (the leaf label in the decision tree from Figure 5) as well as (2) the response groupings. An example fingerprint is ⟨R1, {{NSD, KNOT, POWERDNS, YADIFA}, {BIND, COREDNS}, {TRUSTDNS, MARADNS}}⟩. The fingerprint for an ill-formed test is similar but we use the validity condition being violated instead of the model case.

5 Results

5.1 Testing Using Valid Zone Files

Using FERRET, we generated thousands of tests and used them to compare the behavior of 8 popular open-source authoritative implementations of DNS. Table 3 shows the 8 implementations, the languages they are implemented in, and a brief description of their focus or how they are used. We constrained FERRET to generate tests where the length of each domain name and the number of records in the zone was at most 4. We ran FERRET on a 3.6GHz 72 core machine with 200 GB of RAM and it generated a total of 12,673 valid test cases, one per path in our Zen model that is consistent with the length constraints, in approximately 6 hours. Users can run the tests in parallel, so the runtime depends heavily on the

Implementation	Language	Description
BIND [23]	C	*de facto* standard
POWERDNS [21]	C++	popular in N. Europe
NSD [76]	C	hosts several TLDs
KNOT [25]	C	hosts several TLDs
COREDNS [16]	Go	used in Kubernetes
YADIFA [29]	C	created by EURid (.eu)
TRUSTDNS [33]	Rust	security, safety focused
MARADNS [102]	C	lightweight server

Table 3: The eight open-source DNS nameserver implementations tested by FERRET. FERRET can test implementations implemented in any language.

Model Case	#Tests	#Tests Failing	#Fingerprints
E1	3180	239	7
E2	12	10	5
E3	96	12	3
E4	6036	5312	11
W1	60	33	8
W2	24	21	9
W3	18	16	1
D1	230	65	4
R1	2980	2529	27
R2	37	3	1

Table 4: Test generation statistics for $n = 4$. The model case refers to the leaves in Figure 5. Even though the number of failed tests is higher, the number of fingerprints is small.

user resources for parallelization. Each test takes around 10 seconds to run on average, and most of the time is spent setting up the zone file and necessary configuration files.

As described in § 4, FERRET runs each test against all 8 implementations and groups their responses. Out of 12,673 tests, FERRET found more than one group in the majority (8,240) of tests. Table 4 shows the number of tests generated for each case in the model (Figure 5), the number of tests where there was more than one group, and the number of unique fingerprints formed for each model case.

In total the 8,240 tests with more than one group were partitioned into 76 unique fingerprints, for a reduction of more than two orders of magnitude. For 24 of these fingerprints there exists only a single test case, while one fingerprint has 1892 corresponding tests. These 76 fingerprints can over-count the number of bugs since a single implementation issue can cause errors on multiple model paths. For example, YADIFA, TRUSTDNS, and MARADNS do not support DNAME records; so any generated test containing this feature will cause them to give the wrong answer or fail to respond. However, two tests can also have the same fingerprint despite different implementation root causes; so the number of fingerprints can

also under-count the number of bugs.

For these reasons, we manually examined the test cases matching each fingerprint, examining them all when the fingerprint has 4 or fewer tests and otherwise examining a small random sample. By doing this we identified 24 unique bugs, as summarized in Table 6 (all except the ones marked with ✧). All of these have been confirmed as actual bugs (no false positives) and developers have fixed 14 of them at the time of writing.

5.2 Testing Using Invalid Zone Files

FERRET generated 900 ill-formed zone files, 100 violating each of the validity conditions in Table 2, in 2.5 hours. We used these zone files to test the four most widely used DNS implementations — BIND, NSD, KNOT, POWERDNS— as these have a mature zone-file preprocessor available.

There is no practical limit on the number of invalid zone files the tool can generate. We limited it to 100 for each violation in our experiments, but one could use FERRET to generate many more such tests if desired. Similarly, though we only explored violations of single well-formedness rules, it is straightforward to use FERRET to generate tests that violate a combination of rules. As a first step, FERRET checked all of the zone files with each implementation's preprocessor: named-checkzone [24] for BIND, kzonecheck [18] for KNOT, nsd-checkzone [77] for NSD, and pdnsutil [20] for POWERDNS. Each implementation can either reject or accept the invalid zone file and Table 5 shows the statistics of how different implementations treat the zone files.

All together there are 573 invalid zone files (the first five rows in the table) that are accepted by more than one DNS implementation and so are amenable to differential testing. Our formal model relies on zones to be well-formed: so we cannot use it to generate queries for these zones. Instead we leverage GROOT [71], which generates query equivalence classes (ECs) of the form ⟨example.com, t⟩ for a given zone file, one for each DNS record type t, and does not require the zone to be semantically well-formed. We used 7 query types: A, NS, CNAME, DNAME, SOA, TXT, AAAA. We excluded 19 zone files as GROOT generated over 200 ECs for each of them due to multiple interacting DNAME loops. For the remaining 554 zone files, the average number of ECs is 21*7 i.e., 21 domains names and each domain name is paired with the 7 types, and we chose one representative query from each EC.

The last column in Table 5 shows the results of differential testing. For example, 106 out of the 201 zone files in the first row exhibited differences among the three implementations during testing. We manually inspected all differences for the zone files that violated conditions of i, ii, iii, vi, and ix, as there were 12 or fewer such differences in each category, and we inspected a random sample for the others. By doing this we identified 6 new errors as shown in Table 6 with the ✧ symbol and all of them are fixed. Some of the errors identified earlier were also present here but are not double-counted.

B I N D	N S D	K N O T	P D N S	#Zones	Condition violated	#Zones with a difference
A	A	A	R	100 + 100 + 1	i or viii or ix	11 + 94 + 1
A	A	R	R	100 + 61	vi or ix	8 + 3
A	R	A	R	17 + 100	ii or iii	1 + 6
A	R	R	A	60	vii	53
R	A	R	A	34	ix	7
A	R	R	R	39	vii	-
R	A	R	R	4	ix	-
R	R	R	A	95 + 1	v or vii	-
R	R	R	R	83 + 100 + 5	ii or iv or v	-

Table 5: Invalid zone file statistics. The second row shows that 100 (61) zone files that violate condition vi (ix) are accepted by only BIND and NSD, and 8 (3) of them resulted in some difference between the two implementations.

5.3 Example Bugs

We now provide a detailed description of some of the bugs from Table 6. Two of them were already described in § 2.2.

Bug #3: COREDNS Crash. FERRET generated the following test that causes COREDNS, the recommended nameserver for Kubernetes, to crash. It was subsequently confirmed and fixed by the COREDNS developers.

```
example.    SOA      ...
*.example.  CNAME   foo.example.
```
Query: ⟨baz.bar.example., CNAME⟩

In this example the zone file has a wildcard CNAME record that rewrites any query ending with the label example to foo.example. This rewritten query will then match the wildcard record again and so on, causing COREDNS to loop and consume resources until, eventually, the server crashes with the following message:

```
runtime: goroutine stack exceeds 1000000000-byte limit
runtime: sp=0xc03c6c0378 stack=[0xc03c6c0000, ...]
fatal error: stack overflow
```

Interestingly, COREDNS correctly guards against CNAME loops that do not involve wildcard; so only a test that combines CNAME and wildcards will trigger the bug. After our bug report, the developers fixed the issue by adding a loop counter and breaking the loop if the depth exceeds nine. They commented: "Note the answer we're returning will be incomplete (more cnames to be followed) or illegal (wildcard cname with multiple identical records). For now it's more important to protect ourselves than to give the client a valid answer."

Crashes like this represent serious security vulnerabilities, particularly in multi-tenant settings such as the attack described earlier in Figure 4(a).

Bug #4: Wrong RCODE for synthesized CNAME. FERRET generated a zone that violates condition vii in Table 2:

Implementation	Bugs Found	Bug Type	Status
BIND	Sibling glue records not returned [47]	Wrong Additional	✓
	Zone origin glue records not returned [45]	Wrong Additional	✓
	DNAME recursion denial-of-service$^\diamond$ [44]	Server Crash	☑
	Wrong RCODE for synthesized record$^\diamond$ [46]	Wrong RCODE	☑
NSD	DNAME not applied recursively [65]	Wrong Answer	☑
	Wrong RCODE when * is in Rdata [64]	Wrong RCODE	☑
	Used NS records below delegation$^\diamond$ [67]	Wrong Answer	☑
	Wrong RCODE for synthesized record$^\diamond$ [66]	Wrong RCODE	☑
POWERDNS	CNAME followed when not required [62]	Wrong Answer	✓
	pdnsutil check-zone DNAME-at-apex$^\diamond$ [63]	Preprocessor Bug	☑
KNOT	Incorrect record synthesis [58]	Wrong Answer	☑
	DNAME not applied recursively [61]	Wrong Answer	☑
	Used records below delegation [59]	Wrong Answer	☑
	Error in DNAME-DNAME loop KNOT test [60]	Faulty KNOT Test	☑
	Wrong RCODE for synthesized record$^\diamond$ [91]	Wrong RCODE	☑
COREDNS	NXDOMAIN for existing domain [53]	Wrong RCODE	☑
	Wrong RCODE for CNAME target [55]	Wrong RCODE	☑
	Wildcard CNAME loops & DNAME loops [52]	Server Crash	☑
	Wrong RCODE for synthesized record [57]	Wrong RCODE	☑
	CNAME followed when not required [56]	Wrong Answer	☑
	Sibling glue records not returned [54]	Wrong Additional	✓
YADIFA	CNAME chains not followed [70]	Wrong Answer	☑
	Wrong RCODE for CNAME target [69]	Wrong RCODE	☑
	Used records below delegation [68]	Wrong Answer	☑
MARADNS†	AA flag set for zone cut NS RRs	Wrong Answer	✓
	Used records below delegation	Wrong Answer	✓
TRUSTDNS†	Wildcard match only one label [49]	Wrong Answer	✓
	Used records below delegation [51]	Wrong Answer	✓
	AA flag set for zone cut NS RRs [50]	Wrong Flag	✓
	CNAME loops crash the server [48]	Server Crash	✓

Table 6: Summary of the bugs found by FERRET across the eight implementations. Status column represents whether the developers responded and acknowledged (✓) and also fixed (☑) to the filed bug report. The † symbol denotes implementations with unreported issues due to missing or unimplemented features. The $^\diamond$ symbol denotes the bugs found exclusively using testing with invalid zone files. We reported all the bugs FERRET identified to the respective developers before publishing this paper.

```
      test.com.    SOA     ...
   foo.test.com.   DNAME   bar.test.com.
cs.foo.test.com.   AAAA    1:db8::2:1
```
Query: ⟨www.foo.test.com., CNAME⟩

BIND and POWERDNS accepted the zone file but NSD and KNOT did not. FERRET chose the above query as the representative from the query EC ⟨α.foo.test.com., CNAME⟩ generated by GROOT, where α represents any sequence of labels that does not start with cs. BIND responded with:

```
"rcode NXDOMAIN",
";ANSWER",
"foo.test.com. 500 IN DNAME bar.test.com.",
"www.foo.test.com. 500 IN CNAME www.bar.test.com.",
```

The response from POWERDNS was the same but with a NOERROR RCODE. The RCODE is important as resolvers can use QNAME minimization (RFC 7816 [6]) to wrongly conclude

domain (non-)existence if an incorrect RCODE is returned. However, since the RFCs do not describe this subtle case, the intended behavior is unclear. Since the query is not relevant to the AAAA record, which violates the validity condition, to further investigate this issue we decided to remove that record and check the responses from NSD and KNOT. Both responded with the same response as BIND, leading us to (wrongly) conclude that the issue was with POWERDNS.

To our surprise, after reporting the issue to POWERDNS they responded: "The PowerDNS behavior looks correct to me. Are you sure BIND, NSD, and Knot all return NXDOMAIN on a CNAME query in this context?" BIND and KNOT noticed the issue we filed on POWERDNS's GitHub and fixed the bug almost immediately, even before we filed reports on their repositories. After some back and forth with the NSD developers they concurred saying: "If you are right that the other implementations do this, then we can do that too; that makes less unexpected surprises in packet responses."

Max Length (n)	2	3	4	5
No. of Tests	52	618	12673	646K (51K tested)
Test generation time	10m	40m	6h	14d
No. of Tests Failing	12	224	8240	41173
No. of Fingerprints	9	22	76	115
No. of Bugs	4	14	24	27

Table 7: Results summary for different bounds.

Bug #5: POWERDNS pdnsutil bug. FERRET generated the following test case and POWERDNS returned an incorrect response, exposing a bug in its zone-file preprocessor.

```
    dept.com.      SOA     ...
    dept.com.      DNAME   dept.edu.
host.dept.com.     A       1.1.1.1
```
Query: ⟨host.dept.com., A⟩

The zone file is considered invalid as it violates condition vii in Table 2. nsd-checkzone and kzonecheck preprocessors reject the zone file but named-checkzone and pdnsutil do not raise any errors or warnings and accept the zone file. When queried for the A record, POWERDNS returned this record even though it should have used the DNAME record. POWERDNS has a long-standing open issue about handling DNAME occlusion (records below a DNAME, which should be ignored), and pdnsutil generally gives a warning but did not in this specific case. We filed a bug report for this test and the developers confirmed a bug in pdnsutil when the DNAME is at the apex of the zone. This is now fixed and pdnsutil gives a warning as in other occlusion cases.

5.4 Small-scope Property Validation

Finally, we performed an experiment to validate the small-scope property that justifies our approach — many interesting behaviors can be covered with small tests. We used FERRET to generate valid tests where the length of each domain name and the number of records in the zone were limited to n, for different values of n. Table 7 shows the results. For example, when $n = 2$ there are 52 feasible paths through the model. FERRET generated the corresponding 52 tests in 10 minutes, out of which 12 had more than one group, and these 12 fell into 9 fingerprints. By inspecting those failed tests, we identified 4 unique bugs, which are a subset of the ones identified by our evaluation described in § 5.1, where $n=4$.

Our experiment identifies two distinct forms of small-scope property. First, the DNS query resolution protocol itself, as represented by our logical model, has a small-scope property. In particular, when $n=2$ all leaf nodes in Figure 5 are covered by at least one test, except for the R1 leaf, and all leaf nodes are covered when n is 3 or higher. Hence, although we are restricted to generating small zones, we can still cover all return points in our formal model, each of which represents a distinct RFC behavior.

Second, the DNS nameserver implementations have a small-scope property. In part the fact that we have identified dozens of subtle new errors is evidence that small tests can explore interesting behaviors. The results in Table 7 add further evidence. As we increase the size of n from 2 to 3 to 4, the number of bugs identified goes from 4 to 14 to 24. In the $n=5$ case, FERRET generated over 646K tests and took almost 14 days to finish. The distribution of tests across model cases is similar to the $n=4$ breakdown shown in Table 4, where the majority of tests fall into the E1, E4 and R1 cases. We randomly sampled 50K tests to run from these three cases, according to their proportions. The other cases totalled to around 1000 tests, so we ran all of them. Out of the resulting 115 fingerprints, 50 fingerprints were in common with the fingerprints of $n=4$. We therefore decided to examine the remaining 65 fingerprints to search for new bugs. For these 65 fingerprints, the median number of tests in each fingerprint was 3, and the mode was 1. We found three bugs that we did not find with $n = 4$, but all three bugs were covered by the tests for invalid zones with $n=4$ (§ 5.2). In other words, increasing n from 4 to 5 has so far not uncovered any new errors in the DNS nameserver implementations.

6 Discussion

Our SCALE approach worked surprisingly well at identifying subtle errors in implementations. This was not obvious from the beginning, since each implementation can have very different control logic compared to one another and compared to our formal model. And yet seemingly the tests derived from paths through our formal RFC model frequently uncover bugs in rare control paths for these implementations.

On the other hand, this approach is not a panacea. We found situations where one path in the model corresponds to multiple paths in an implementation due to the internal data structures that it uses to represent different record types, which can lead to FERRET missing some issues. This showed up, for example, with empty non-terminals (ENTs) – domain names that own no resource records but have subdomains that do. Since there is no explicit branch that differentiates empty non-terminals in the model, FERRET did not generate test cases where the zone file had both an ENT and a query targeting that ENT. However, by manually testing a few such cases, we found two more bugs in COREDNS. Going forward it may be possible to extend FERRET to find more cases like this by adding additional non-semantic branches to the model to expose behavior thought to be error-prone.

More generally, we believe our SCALE approach to RFC compliance testing and "ferreting" out bugs through (i) symbolic execution of a small formal model to jointly generate configurations together with inputs, combined with (ii) differential testing, and (iii) fingerprinting, could be useful more broadly beyond the DNS. For instance, there are many other complex and distributed protocols used at different network layers such as routing protocols like BGP and OSPF,

flow control protocols like PFC, new transport layer protocols such as QUIC, and many more. It would be interesting future work to apply the SCALE methodology beyond DNS.

7 Related Work

FERRET and SCALE are related to several lines of prior work in DNS and in automated testing.

Verified DNS implementations. One approach is to build, from scratch, a nameserver implementation verified to be correct. This approach has found some success in other domains, for example, in operating system microkernels [73] using proof assistants such as Coq [79]. Ironsides [12] is an implementation of a DNS resolver and authoritative nameserver that uses SPARK [3] to prove the absence of dataflow errors such as buffer overflows. While this work is promising, it does not formalize the DNS RFC semantics and thus cannot provide any functional correctness guarantees. Moreover, open source implementations such as BIND [23] are already used pervasively in the Internet. Providing a new verified implementation does not help these existing deployments.

Models for DNS. In our prior work on the GROOT zone-file verifier [71] we provided the first formalization of DNS semantics. However, it was a paper formalism and was only used to prove the correctness of the equivalence-class generation algorithm that forms the core of GROOT's approach to verifying zone files. Indeed, GROOT assumes that DNS implementations conform to the DNS RFCs. Our work is therefore complementary, but we used GROOT's logical model as a basis for our executable Zen model. We also leveraged GROOT's equivalence-class generation algorithm to create queries for invalid zone files.

Fuzz testing. Fuzz testing with semi-random and/or grammar-based tests has seen success in recent years [1, 5, 40, 78, 101]. However, as mentioned in § 1, fuzzing cannot easily be used in our setting due to the need to navigate complex constraints and dependencies, and hence existing fuzzers for DNS [10, 89, 99] are limited to testing DNS parsers and use a fixed zone file.

Symbolic execution. Symbolic execution [36, 37], which systematically solves for inputs that take different execution paths in a program, has also been successful [9, 11]. However, as described in § 1, due to the scale and complexity of DNS nameserver implementations, symbolic execution has been used only on individual functions and has avoided the need to generate zone files [93]. Our SCALE approach uses symbolic execution to drive test generation, but it does so on an executable model of the RFC behavior, which is significantly smaller and simpler than an implementation and has carefully chosen data representations that are amenable to symbolic execution. As a result, symbolic execution on our model is tractable and allows us to jointly generate (small) zone files and DNS queries that exercise interesting behaviors.

Model- and specification-based testing. In model-based testing (MBT) [8, 88, 90, 104] a user builds an abstract model of the system to test (e.g., a finite state machine [8, 104]). A tester implementation then generates paths through this abstract model and creates concrete tests by "filling in" missing information from the abstract example. Closest to our work are model-based testers for black-box network functions (e.g., [30, 98]), which also use symbolic execution to generate tests. However, they respectively use finite-state machine models [30] and a domain-specific language for specifying network function behavior [98], while we have implemented a full functional model of DNS in a general modeling language [4]. Further, their setting does not require generating configurations, which is the key technical challenge for testing protocols like DNS.

Specification-based testing leverages a user-provided specification of the valid inputs to a function. Most commonly, tests are generated by finding inputs that satisfy a given precondition [7]. Like SCALE these approaches typically rely on a small-scope hypothesis [43] and hence focus on generating small inputs. Recent work has developed an approach to automated testing for QUIC implementations [81, 82] that leverages a formal specification, but in a very different way than in our approach. Specifically, the specification models the party that is interacting with the implementation being tested and is used to generate valid responses.

Finally, recent works automatically learn protocol models from implementations [31] or RFCs [105]. We could potentially adopt these techniques in the future to reduce the burden of producing our formal model.

8 Conclusion

Despite its importance as the "phonebook" of the Internet, DNS is fraught with implementation bugs that can impact millions of users. In this paper, we introduced FERRET, the first automatic test generator for RFC compliance of DNS nameserver implementations. The SCALE approach underlying FERRET uses symbolic execution of a formal model to jointly generate configurations together with inputs. FERRET combines this technique with differential testing and fingerprinting to identify and automatically triage implementation errors. In total FERRET identified 30 new bugs, including at least two for each of the 8 implementations that we tested. We believe that this combination of techniques can generalize to "ferret" out subtle RFC-compliance bugs in large implementation code bases for other network protocols that use configurations.

Acknowledgements

References

[1] American Fuzzing Lop (AFL). Afl 2018.
https://lcamtuf.coredump.cx/afl/.

[2] Amazon. Route 53.
https://aws.amazon.com/route53/.

[3] John Barnes. *Spark: The Proven Approach to High Integrity Software*. Altran Praxis, London, GBR, 2012.

[4] Ryan Beckett and Ratul Mahajan. A general framework for compositional network modeling. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, HotNets '20, page 8–15, New York, NY, USA, 2020. Association for Computing Machinery.

[5] Marcel Böhme, Van-Thuan Pham, and Abhik Roy-choudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1032–1043, New York, NY, USA, 2016. Association for Computing Machinery.

[6] Stéphane Bortzmeyer. DNS Query Name Minimisation to Improve Privacy. RFC 7816, March 2016.

[7] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on java predicates. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, page 123–133, New York, NY, USA, 2002. Association for Computing Machinery.

[8] Josip Bozic, Lina Marsso, Radu Mateescu, and Franz Wotawa. A formal tls handshake model in lnt. In John P. Gallagher, Rob van Glabbeek, and Wendelin Serwe, editors, Proceedings Third Workshop on *Models for Formal Analysis of Real Systems* and Sixth International Workshop on *Verification and Program Transformation*, Thessaloniki, Greece, 20th April 2018, volume 268 of *Electronic Proceedings in Theoretical Computer Science*, pages 1–40, Greece, 2018. Open Publishing Association.

[9] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 209–224, USA, 2008. USENIX Association.

[10] Frederic Cambus. Fuzzing dns zone parsers.
https://www.cambus.net/fuzzing-dns-zone-parsers/.

[11] Marco Canini, Vojin Jovanović, Daniele Venzano, Dejan Novaković, and Dejan Kostić. Online testing of federated and heterogeneous distributed systems. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, page 434–435, New York, NY, USA, 2011. Association for Computing Machinery.

[12] M. Carlisle and B. Fagin. Ironsides: Dns with no single-packet denial of service or remote code execution vulnerabilities. In *2012 IEEE Global Communications Conference (GLOBECOM)*, pages 839–844, Anaheim, CA, USA, 2012. IEE.

[13] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725, 2018.

[14] Taolue Chen, Yan Chen, Matthew Hague, Anthony W. Lin, and Zhilin Wu. What is decidable about string constraints with the replaceall function. *Proc. ACM Program. Lang.*, 2(POPL), December 2017.

[15] Bind Community. Bind gitlab issues.
https://gitlab.isc.org/isc-projects/bind9/-/issues.

[16] CoreDNS community. Coredns.
https://coredns.io/.
Code commit used: https://github.com/coredns/coredns/tree/6edc8fe7f6c2f57844c8ee7f7f5deef71085ebe8.

[17] Dnspython Community. Dnspython.
https://dnspython.readthedocs.io/en/latest/index.html.

[18] Knot community. kzonecheck – knot dns zone file checking tool.
https://www.knot-dns.cz/docs/2.5/html/man_kzonecheck.html.

[19] NSD Community. Nsd github issues.
https://github.com/NLnetLabs/nsd/issues.

[20] PowerDNS community. Pdnsutil.
https://doc.powerdns.com/authoritative/manpages/pdnsutil.1.html.

[21] PowerDNS Community. Powerdns.
https://www.powerdns.com/.
Code commit used: https://github.com/PowerDNS/pdns/tree/a03aaad7554483ee6efe72a81eda00a9d1a94fe5.

[22] PowerDNS Community. Powerdns github issues.
https://github.com/PowerDNS/pdns/issues?q=is%3Aissue+is%3Aopen+label%3Aauth.

[23] Internet Systems Consortium. Bind 9.
https://www.isc.org/bind/.
Code commit used: https://gitlab.isc.org/isc-projects/bind9/-/tree/dbcf683c1a57f49876e329fca183cb39d20ca3a4.

[24] Internet Systems Consortium. named-checkzone(8).
https://linux.die.net/man/8/named-checkzone.

[25] CZ.NIC. Knot.
https://www.knot-dns.cz/.
Code commit used: https://gitlab.nic.cz/knot/knot-dns/-/tree/563fcdd886b5d5c52bceeb8fda3c4bda59ece73e.

[26] National Vulnerability Database. CVE-2021-25215 Detail.
https://nvd.nist.gov/vuln/detail/CVE-2021-25215.

[27] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[28] Robert Elz and Randy Bush. Clarifications to the DNS Specification. RFC 2181, July 1997.

[29] EURid.eu. Yadifa.
https://www.yadifa.eu/.
Code commit used: https://github.com/yadifa/yadifa/tree/dc5bed2fb8ec204af9b65eeb91934c2c85098cbb.

[30] Seyed K. Fayaz, Tianlong Yu, Yoshiaki Tobioka, Sagar Chaki, and Vyas Sekar. BUZZ: Testing Context-Dependent policies in stateful networks. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 275–289, Santa Clara, CA, March 2016. USENIX Association.

[31] Tiago Ferreira, Harrison Brewton, Loris D'Antoni, and Alexandra Silva. Prognosis: Closed-box analysis of network protocol implementations. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 762–774, New York, NY, USA, 2021. Association for Computing Machinery.

[32] Jonathan Foote. How to fuzz a server with american fuzzy lop.
https://www.fastly.com/blog/how-fuzz-server-american-fuzzy-lop, 2015.

[33] Benjamin Fry and Community. Trust-dns.
http://trust-dns.org/.
Code commit used: https://github.com/bluejekyll/trust-dns/tree/7d9b186121fb5cb331cf2ec6baa47846b83de8fc.

[34] James Fryman. Dns outage post mortem.
https://github.blog/2014-01-18-dns-outage-post-mortem/, 2014.

[35] Vijay Ganesh, Mia Minnes, Armando Solar-Lezama, and Martin Rinard. Word equations with length constraints: What's decidable? In *Proceedings of the 8th International Conference on Hardware and Software: Verification and Testing*, HVC'12, page 209–226, Berlin, Heidelberg, 2012. Springer-Verlag.

[36] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, page 206–215, New York, NY, USA, 2008. Association for Computing Machinery.

[37] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, page 213–223, New York, NY, USA, 2005. Association for Computing Machinery.

[38] Suzanne Goldlust, Michał Kępień, Peter Davies, and Everett Fulton. CVE-2021-25215: An assertion check can fail while answering queries for DNAME records that require the DNAME to be processed to resolve itself.
https://kb.isc.org/v1/docs/cve-2021-25215.

[39] Google. Cloud dns.
https://cloud.google.com/dns.

[40] Sam Hocevar. zzuf: multi-purpose fuzzer.
http://caca.zoy.org/wiki/zzuf/., 2007.

[41] Paul E. Hoffman, Andrew Sullivan, and Kazunori Fujiwara. DNS Terminology. RFC 8499, January 2019.

[42] Dyn Inc. Dynamic dns.
https://account.dyn.com/.

[43] Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, April 2002.

[44] Siva Kakarla, Mark Andrews, Michał Kępień, Peter Davies, and Michal Nowak. [CVE-2021-25215] An assertion check can fail while answering queries for DNAME records that require the DNAME to be processed to resolve itself.
https://gitlab.isc.org/isc-projects/bind9/-/issues/2540.

[45] Siva Kesava R Kakarla and Mark Andrews. Glue records can be returned when the name server's name is same as the zone origin.
https://gitlab.isc.org/isc-projects/bind9/-/issues/2385.

[46] Siva Kesava R Kakarla, Mark Andrews, and Michał Kępień. DNAME: synthetized CNAME might be perfect answer to CNAME query.
https://gitlab.isc.org/isc-projects/bind9/-/issues/2284.

[47] Siva Kesava R Kakarla, Mark Andrews, and Michał Kępień. Sibling (In-bailiwick rule of RFC 8499) domain IP records not returned.
https://gitlab.isc.org/isc-projects/bind9/-/issues/2384.

[48] Siva Kesava R Kakarla and Benjamin Fry. CNAME loops throws off the server.
https://github.com/bluejekyll/trust-dns/issues/1283.

[49] Siva Kesava R Kakarla and Benjamin Fry. Wildcards match only one label.
https://github.com/bluejekyll/trust-dns/issues/1342.

[50] Siva Kesava R Kakarla and Benjamin Fry. Zone cut NS RRs returned as authoritative records.
https://github.com/bluejekyll/trust-dns/issues/1273.

[51] Siva Kesava R Kakarla, Benjamin Fry, and Jonas Bushart. Glue records returned as authoritative records by the server .
https://github.com/bluejekyll/trust-dns/issues/1272.

[52] Siva Kesava R Kakarla and Miek Gieben. Handling wildcard CNAME loops.
https://github.com/coredns/coredns/issues/4378.

[53] Siva Kesava R Kakarla and Miek Gieben. NXDO-MAIN returned when the domain exists.
https://github.com/coredns/coredns/issues/4374.

[54] Siva Kesava R Kakarla and Miek Gieben. Sibling (In-bailiwick rule of RFC 8499) domain IP records can also be returned along with NS records.
https://github.com/coredns/coredns/issues/4377.

[55] Siva Kesava R Kakarla and Chris O'Haver. Non-existent CNAME target in the same zone should be returned with NXDOMAIN instead of NOERROR rcode.

[56] Siva Kesava R Kakarla, Chris O'Haver, and Kohei Yoshida. CNAME need not be followed after a synthesized CNAME for a CNAME query.
https://github.com/coredns/coredns/issues/4398.

[57] Siva Kesava R Kakarla, Chris O'Haver, and Kohei Yoshida. Return code for synthesized CNAME records (from wildcards and DNAMEs).
https://github.com/coredns/coredns/issues/4341.

[58] Siva Kesava R Kakarla, Libor Peltan, and Daniel Salzman. Record incorrectly synthesized from wildcard record.
https://gitlab.nic.cz/knot/knot-dns/-/issues/715.

[59] Siva Kesava R Kakarla, Libor Peltan, and Daniel Salzman. Records below delegation are not ignored (kzonecheck also does not raise any issue).
https://gitlab.nic.cz/knot/knot-dns/-/issues/713.

[60] Siva Kesava R Kakarla, Libor Peltan, Daniel Salzman, and mscbg. DNAME-DNAME loop test case is not a loop.
https://gitlab.nic.cz/knot/knot-dns/-/issues/703.

[61] Siva Kesava R Kakarla, Libor Peltan, Daniel Salzman, and Vladimír Čunát. DNAME not applied more than once to resolve the query.
https://gitlab.nic.cz/knot/knot-dns/-/issues/714.

[62] Siva Kesava R Kakarla and Peter van Dijk. CNAME need not be followed after a synthesized CNAME for a CNAME query.
https://github.com/PowerDNS/pdns/issues/9886.

[63] Siva Kesava R Kakarla and Peter van Dijk. pdnsutil DNAME checks have issues.
https://github.com/PowerDNS/pdns/issues/9734.

[64] Siva Kesava R Kakarla and Wouter Wijngaards. '*' in Rdata causes the return code to be NOERROR instead of NX.
https://github.com/NLnetLabs/nsd/issues/152.

[65] Siva Kesava R Kakarla and Wouter Wijngaards. DNAME not applied more than once to resolve the query.
https://github.com/NLnetLabs/nsd/issues/151.

[66] Siva Kesava R Kakarla and Wouter Wijngaards. DNAME: synthesized CNAME might be perfect answer to CNAME query.
https://github.com/NLnetLabs/nsd/issues/140.

[67] Siva Kesava R Kakarla and Wouter Wijngaards. NS Records below delegation are not ignored (nsd-checkzone also does not raise any issue).
https://github.com/NLnetLabs/nsd/issues/174.

[68] Siva Kesava R Kakarla and yadifa. Records below delegation are not ignored.
https://github.com/yadifa/yadifa/issues/12.

[69] Siva Kesava R Kakarla, yadifa, and edfeu. Non-existent CNAME target in the same zone should be returned with NXDOMAIN instead of NOERROR.
https://github.com/yadifa/yadifa/issues/11.

[70] Siva Kesava R Kakarla, yadifa, and edfeu. Why are CNAME chains not followed?
https://github.com/yadifa/yadifa/issues/10.

[71] Siva Kesava Reddy Kakarla, Ryan Beckett, Behnaz Arzani, Todd Millstein, and George Varghese. Groot: Proactive verification of dns configurations. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 310–328, New York, NY, USA, 2020. Association for Computing Machinery.

[72] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[73] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. Sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery.

[74] Eduard Kovacs. Dns servers crash due to bind security flaw.
https://www.securityweek.com/dns-servers-crash-due-bind-security-flaw, 2018.

[75] Marc Kührer, Thomas Hupperich, Jonas Bushart, Christian Rossow, and Thorsten Holz. Going wild: Large-scale classification of open dns resolvers. In *Proceedings of the 2015 Internet Measurement Conference*, IMC '15, page 355–368, New York, NY, USA, 2015. Association for Computing Machinery.

[76] NLnet Labs. Nsd.
https://nlnetlabs.nl/projects/nsd/about/. Code commit used: https://github.com/NLnetLabs/nsd/tree/4043a5ab7be7abaec969011e48e4d0d60a0056a6.

[77] NLnet Labs. nsd-checkzone - nsd zone file syntax checker.
https://www.nlnetlabs.nl/documentation/nsd/nsd-checkzone/.

[78] Hyojeong Lee, Jeff Seibert, Dylan Fistrovic, Charles Killian, and Cristina Nita-Rotaru. Gatling: Automatic performance attack discovery in large-scale distributed systems. *ACM Trans. Inf. Syst. Secur.*, 17(4), April 2015.

[79] Pierre Letouzey. *Programmation fonctionnelle certifiée: l'extraction de programmes dans l'assistant Coq*. PhD thesis, Université Paris Sud, 2004.

[80] Edward P. Lewis. The Role of Wildcards in the Domain Name System. RFC 4592, July 2006.

[81] Kenneth L. McMillan and Lenore D. Zuck. Compositional testing of internet protocols. In *2019 IEEE Cybersecurity Development (SecDev)*, pages 161–174, 2019.

[82] Kenneth L. McMillan and Lenore D. Zuck. Formal specification and testing of quic. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 227–240, New York, NY, USA, 2019. Association for Computing Machinery.

[83] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239):2, March 2014.

[84] Microsoft. Github, inc.
https://github.com/.

[85] Microsoft. Microsoft dns.
https://en.wikipedia.org/wiki/Microsoft_DNS.

[86] P. Mockapetris. Domain names - concepts and facilities. RFC 1034, November 1987.

[87] Paul Mockapetris. Domain names - implementation and specification. RFC 1035, November 1987.

[88] B. Neelakantan and S. V. Raghavan. *Protocol Conformance Testing — A Survey*, pages 175–191. Springer US, Boston, MA, 1995.

[89] NMAP Organization. Dns-fuzz.
https://nmap.org/nsedoc/scripts/dns-fuzz.html.

[90] Javier Paris and Thomas Arts. Automatic testing of tcp/ip implementations using quickcheck. In *Proceedings of the 8th ACM SIGPLAN Workshop on ERLANG*, ERLANG '09, page 83–92, New York, NY, USA, 2009. Association for Computing Machinery.

[91] Libor Peltan and Daniel Salzman. DNAME: synthesized CNAME might be perfect answer to CNAME query.
https://gitlab.nic.cz/knot/knot-dns/-/merge_requests/1217.

[92] Libor Peltans. Nsd and knot discussion.
https://github.com/NLnetLabs/nsd/issues/142#issuecomment-732753256.

[93] David A. Ramos and Dawson Engler. Under-constrained symbolic execution: Correctness checking for real code. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 49–64, Washington, D.C., August 2015. USENIX Association.

[94] Fahmida Y. Rashid. Isc updates critical dos bug in bind dns software.
https://www.infoworld.com/article/3126472/isc-updates-critical-dos-bug-in-bind-dns-software.html, 2016.

[95] Andrew Reynolds, Jasmin Christian Blanchette, Simon Cruanes, and Cesare Tinelli. Model finding for recursive functions in smt. In Nicola Olivetti and Ashish Tiwari, editors, *Automated Reasoning*, pages 133–151, Cham, 2016. Springer International Publishing.

[96] Scott Rose and Wouter Wijngaards. DNAME Redirection in the DNS. RFC 6672, June 2012.

[97] Kyle Schomp, Onkar Bhardwaj, Eymen Kurdoglu, Mashooq Muhaimen, and Ramesh K. Sitaraman. Akamai dns: Providing authoritative answers to the world's queries. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 465–478, New York, NY, USA, 2020. Association for Computing Machinery.

[98] Harsha Sharma, Wenfei Wu, and Bangwen Deng. Symbolic execution for network functions with time-driven

logic. In *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 1–8, 2020.

[99] Robert Swiecki and *et al.* Honggfuzz - security oriented software fuzzer.
https://github.com/google/honggfuzz/tree/master/examples/bind.

[100] Dmitriy Zaporozhets Sytse "Sid" Sijbrandij. Gitlab, inc.
https://gitlab.com/.

[101] Peach Tech. Peach fuzzer platform.
peach.tech/products/peach-fuzzer/peach.tech/products/peach-fuzzer/.

[102] Sam Trenholme. Maradns.
https://maradns.samiam.org/.
Code commit used: https://github.com/samboy/MaraDNS/tree/3ec477f227b2bf6947be8fbe8fd0ab73130227d0.

[103] Liam Tung. Azure global outage: Our dns update mangled domain records, says microsoft.
https://www.zdnet.com/article/azure-global-outage-our-dns-update-mangled-domain-records-says-microsoft/, 2019.

[104] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. *Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer*, pages 39–76. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[105] Jane Yen, Tamás Lévai, Qinyuan Ye, Xiang Ren, Ramesh Govindan, and Barath Raghavan. Semi-automated protocol disambiguation and code generation. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 272–286, New York, NY, USA, 2021. Association for Computing Machinery.

[106] Dan York. Hbo now dnssec misconfiguration makes site unavailable from comcast networks (fixed now).
https://www.internetsociety.org/blog/2015/03/hbo-now-dnssec-misconfiguration-makes-site-unavailable-from-comcast-networks-fixed-now/.

Decentralized cloud wide-area network traffic engineering with BLASTSHIELD

Umesh Krishnaswamy   Rachee Singh   Nikolaj Bjørner   Himanshu Raj

Microsoft

Abstract

Cloud networks are increasingly managed by centralized software defined controllers. Centralized traffic engineering controllers achieve higher network throughput than decentralized implementations, but are a single point of failure in the network. Large scale networks require controllers with isolated fault domains to contain the *blast radius* of faults. In this work, we present BLASTSHIELD, Microsoft's software-defined decentralized WAN traffic engineering system. BLASTSHIELD *slices* the WAN into smaller fault domains, each managed by its own slice controller. Slice controllers independently engineer traffic in their slices to maximize global network throughput without relying on hierarchical or central coordination. BLASTSHIELD is fully deployed in Microsoft's WAN and carries a majority of the backbone traffic. BLASTSHIELD achieves similar network throughput as the previous generation centralized controller and reduces traffic loss from controller failures by 60%.

1   Introduction

Cloud wide-area networks (WANs) enable low-latency and high bandwidth cloud applications like live-video, geo-replication, and other business critical workloads. Cloud WANs are billion-dollar assets, and annually cost a hundred million dollars to maintain. To efficiently utilize their infrastructure investment, cloud providers employ centralized, software-defined traffic engineering (TE) systems. Centralized TE leverages global views of the topology and demands to maximize the network throughput.

Maximum throughput, but at what cost? The paradigm shift in WAN TE from fully decentralized switch-native protocols (*e.g.,* RSVP-TE [4]) to centralized TE controllers was driven by the throughput gains made possible by centralization [16]. After a decade of operating the software-defined WAN (SWAN) in Microsoft's backbone network, we claim that it is more important that the centralized TE controller does not become a single point of failure in the system. The impact of a TE controller fault needs to be lowered along with achieving high throughput.

Controller replication does not guarantee availability. Our operational experience with SWAN has taught us that regardless of good engineering practices (*e.g.,* code reviews, safe deployment, testing and verification), software systems will fail

in production in unforeseen ways, often due to complex interactions of multiple faults. While it is hard to eliminate faults, it is crucial to contain the damage when faults inevitably occur. Despite fault-tolerant components of the SWAN TE system and replication of the centralized TE controller, an unforeseen cascade of faults led to an outage of global scope in the SWAN TE system.

In this work, we first describe the operational experiences that led us to migrate away from SWAN, the fully centralized TE system in the Microsoft cloud network (§ 2). Second, to reason about the availability of large-scale wide-area TE systems, we define *blast radius* of a TE controller as the fraction of customer or tier-0 traffic at risk due to its failure. We developed BLASTSHIELD, a WAN TE system that reduces the blast radius by slicing the global cloud WAN into smaller fault domains or *slices* (§ 3). BLASTSHIELD dials back from fully centralized to slice-decentralized TE by striking a balance between the centralized vs. distributed design principles.

BLASTSHIELD slices are independent, and do not rely on hierarchical or central coordination. Multiple WAN slices and controllers raise unique implementation challenges for BLASTSHIELD. In SWAN, a centralized controller with global view of the network, programmed TE routes in all WAN routers. In contrast, BLASTSHIELD slice controllers work independently — each with its own version of code, configuration, and view of the global network topology. Inconsistent views of the network topology can cause routing loops for inter-slice traffic in the cloud WAN. The failure of a slice controller on the path could blackhole traffic. BLASTSHIELD solves these challenges by developing a robust inter-slice routing mechanism that falls back on switch-native protocol routes in case of slice controller failures (§ 4 and § 5).

We have been operating Microsoft's backbone with BLAST-SHIELD since 2020. We find that BLASTSHIELD allows us to deploy changes to the network safely without the risk of global impact. While any change in network configuration or software is accompanied by risk, the ability to deploy changes without global risk is a significant advantage. Quantitatively, BLASTSHIELD reduces the risk of traffic loss due to failure of a TE controller by 60%, compared to SWAN (§ 6).

2   Background and Motivation

In this section, we describe an outage in the SWAN network that motivated the design of BLASTSHIELD. This outage was caused by a cascade of several independent failures and its

ripple effects persisted long after the root cause was resolved. The experience of resolving this incident urged us to survey the components at risk in SWAN and mechanisms to mitigate the risks. We define metrics to quantify the availability of TE controllers and design a TE system robust to global-scale outages like the one SWAN experienced.

2.1 Bad luck comes in threes

Prior to the development of BLASTSHIELD, a series of three unfortunate events occurred causing a SWAN outage of global scope. Global SWAN outages lasting more than a few minutes result in loss of several terabytes of network traffic, and are instantly observed by a global audience.

Controller removes all routes. A partially failed web request triggered the first bug that led the SWAN controller to remove all its TE routes from WAN routers. In the absence of controller routes, the traffic gets routed over shortest paths computed by the IGP [18]. This type of fallback is acceptable at a small scale, but not as a network-wide replacement.

Incorrect IGP shortest paths. Second, there were two links with misconfigured IGP link weights. The misconfiguration was inconsequential while the controller routes were present. When the controller removed its routes, these links incorrectly became a part of many shortest paths, consequently attracting more traffic than their capacity.

Delayed controller response time. An automatic recovery process could have restored the controller routes in 3 minutes, but a second controller bug incorrectly assumed that the recovering routers were undergoing maintenance, and held back programming routes on them. The longer recovery caused some internal workloads to dynamically change their traffic class to a higher tier, worsening the load and congestion in the network. The combination of these three cascading faults amplified the amount of traffic affected by the outage.

With the luxury of hindsight, we extract three key lessons from the SWAN incident:

1. **All changes have risk.** Global changes are antithetical to the availability of large-scale systems. We need an ability to gradually deploy changes, starting with staging which are production-like but without real customers, to low impact, and finally high impact regions. Global centralized TE precludes piece-wise rollout of changes.
2. **Configuration and software bugs are inevitable.** The outage occurred due to configuration and software bugs that escaped sandbox validation. While validation can be effective, it remains inherently best effort. In a nutshell, critical infrastructure like SWAN should not presume perfect pre-deployment validation.
3. **Global optimization does not preclude multiple controllers.** In the scenario, non-leader replicas of the controller had an accurate view of the network, and could have

optimized traffic correctly. By partitioning the scope of TE controllers, a faulty leader in one region of the WAN would not impact controllers in other regions.

2.2 Blast Radius, Ripple and Shielding

While faults and small-scale outages occur and get rectified rapidly in our network, what stood out about the SWAN outage incident was its global scope. We define the following terms to quantify the scope of wide-area traffic engineering outages. In later sections, we use these terms to evaluate the reduction in the scope of potential outages when we deploy the new TE system, BLASTSHIELD.

Definition 1 (Blast Radius) *is the fraction of customer or tier-0 traffic at risk by a TE controller failure.*

The service level objective (SLO) is the daily average of the hourly percentage of successfully transmitted bytes. Customer or tier-0 traffic has the highest SLO of 99.999%. Discretionary traffic tiers, tier-1 and tier-2, have a lower SLO of 99.9%. Half the traffic in our network is tier-0. The TE controller routes traffic on engineered paths to optimize for congestion, latency, and diversity. When a TE controller fails by withdrawing its routes or programming incorrect routes or stops programming the network, the ensuing tier-0 loss is the blast radius of the controller.

Definition 2 (Blast Ripple) *of a controller failure is the service level degradation experienced by components that are not governed by the failing TE controller.*

The blast or failure of a TE controller can cause *ripples* and impact traffic not managed by the failing controller. The impact of the ripple is proportional to the amount of tier-0 traffic affected that is not managed by the failing controller.

Definition 3 (Blast Shielding) *is the engineering practice that minimizes the blast radius of failing components while meeting operational constraints like cost and complexity.*

We note that blast shielding does not ensure that the overall system is fault tolerant in achieving the service level objective. Fault tolerance allows the system to operate even if its components fail [3]. Table 1 covers mitigation in Microsoft's TE deployment to achieve fault tolerance and blast shielding. We highlight faults that were not addressed in SWAN's original design and are a focus of this work with ⛊.

3 Slicing the cloud WAN

The global scope of the SWAN outage inspired the design of BLASTSHIELD, the WAN traffic engineering system that has replaced SWAN in Microsoft's backbone network. BLASTSHIELD views the WAN as a collection of sites. Each site

Fault	Mitigation
Controller hardware, cluster, or site failure.	Automatic migration to geo-redundant cluster.
Network fault, *e.g.,* link failure, forwarding fault, router reboot.	Per-router agents perform local repair autonomously without controller intervention. Controller does global repair in the next TE iteration.
Network device disconnects or is unreachable by controller.	Router agents retain last programming. Controller reconnects via router management plane. Router is treated as down if failure persists. Rollback routes if disconnection is during new route programming.
Invalid, inconsistent, outdated programming by controller.	Router agents perform data plane verification. Controller programs agents with latest inputs every 3 minutes.
TE optimization failure *e.g.,* a controller withdraws its routes, or programs incorrect routes. 🛡	Divide the WAN into subgraphs with a controller per subgraph managing a small fault domain.
Malicious router agent *e.g.,* agent stalls the controller from programming other routers. 🛡	Decrease agent-controller interaction to defined subgraphs of the network.
Byzantine controller fault, *e.g.,* a controller sabotages other controllers. 🛡	Controllers acquire network inputs independently.
Zero-day fault in multiple controllers. 🛡	Diverge configurations in TE controllers.

Table 1: Fault types and their mitigation. New fault types handled by this paper are marked with 🛡.
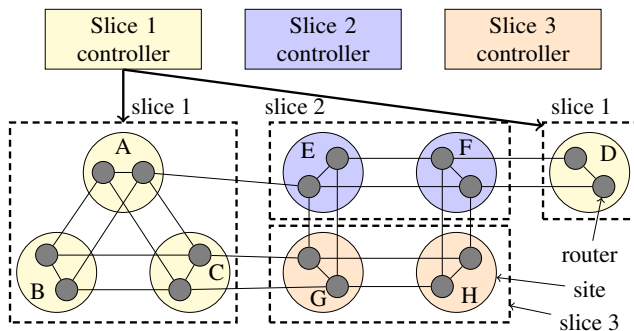

Figure 1: The WAN is divided into slices. Each slice is managed by a dedicated slice controller. Slice 1 consists of routers in sites A–D, slices 2 and 3 have routers in sites E–F and G–H.

consists of multiple WAN routers. WAN routers connect to other routers in the network like the datacenter fabric with a high bandwidth interconnect. WAN routers also transit traffic that is not from a directly connected datacenter. WAN sites at submarine landing terminals and optical transit sites do not have datacenters attached to them.

WAN Slices. BLASTSHIELD divides the WAN into *slices* or subgraphs of routers, each controlled by a dedicated slice controller. A slice is a logical partitioning of the WAN into disjoint sets of routers where each router belongs to exactly one slice. A slice can consist of a single router or all routers, or anything in between. Routers do not have any slice-specific configuration. In Fig. 1, slice 1 consists of routers in sites A–D. A slice can have multiple strongly connected components of routers. Slice 1 has two strongly connected components, the routers in sites A–C and D, respectively. Controllers 2 and 3 manage routers in sites E–F and G–H, respectively. The count and composition of slices is not limited by the design but dictated by operational choice.

Enforcing slice isolation. Only the slice's owning controller

programs routers in the slice. All traffic from slice routers to any destination is engineered by the slice controller. This includes traffic that originates in datacenters directly connected to slice routers and the traffic originating in upstream slice routers. Each slice is a separate deployment and can be patched independently. Slices can inherit common configuration but BLASTSHIELD applies slice-specific configuration independently. Slice controllers do not communicate with another slice controller. This further isolates faults and prevents byzantine controllers bringing the entire system down. Slice controllers operate with a global view of the network by acquiring global topology and demand inputs. Each slice controller makes traffic engineering decisions based on expected conditions in local and remote slices. Controllers anticipate what other controllers do given the same inputs. While deviations between flow allocations computed by different controllers are possible, they are not disruptive to BLAST-SHIELD's operation.

How many slices? The number of BLASTSHIELD WAN slices decide the system's operating point on an important tradeoff between network throughput and blast radius. A single slice enables the TE formulation to achieve maximum network throughput through centralization, but exposes the network to the risk of global blast radius. In contrast, several BLASTSHIELD slices reduce the blast radius of slice controllers but may also reduce the achievable network throughput. Additionally, several WAN slices increase the operational overhead of configuring and maintaining slice controllers. There is a sweet spot for the number of slices that limits the risk of changes and keeps operational overhead manageable. We empirically derive the number of BLASTSHIELD slices for Microsoft's network and strike a balance between blast radius and network throughput (§ 6).

4 BLASTSHIELD System Design

In this section we present the design of BLASTSHIELD and describe the design choices that motivated our design.

4.1 System overview

Each BLASTSHIELD slice controller is a collection of four services: topology service, demand predictor, traffic engineering scheduler, and route programmer (Fig. 2). In addition to the controller services that run on off-router compute nodes, a router agent runs on all WAN routers.
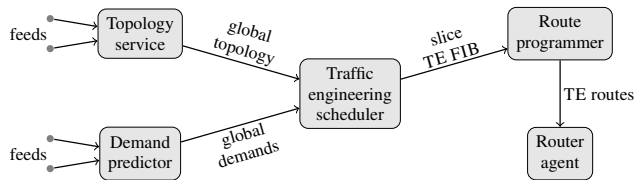

Figure 2: The slice controller consists of topology service, demand predictor, traffic engineering scheduler, and route programmer. Together, they compute traffic engineering routes and program slice routers through router agents.
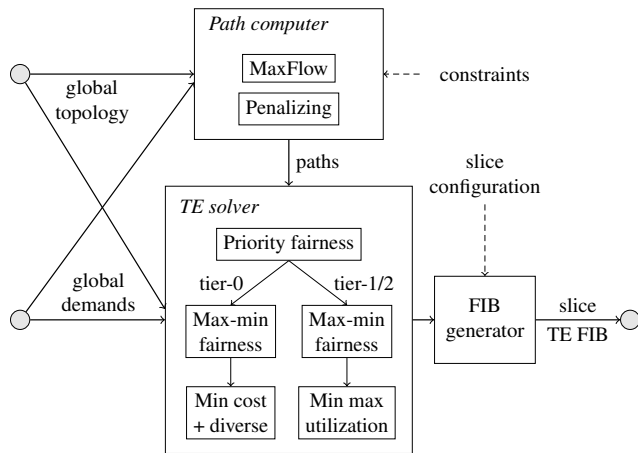

Figure 3: Traffic engineering scheduler computes routes that optimize paths for flows by traffic tier. Each controller performs global optimization based on its view of the entire network, but only programs routers belonging to its slice.

Topology Service synthesizes the global network topology using graph metadata, link state, and router agent input feeds. Graph metadata consists of routers, links, and sites. BGP-LS [15] is the primary source of dynamic link state information *e.g.,* link bandwidths, interface addresses, and segment identifiers [11]. The router agent feed is only used to acquire the health of the router agent; a router must have a functioning agent to be used for traffic engineering.

Demand Predictor predicts upcoming global network demands using a real-time traffic matrices measured by sFlow [26] and host-level packet counters. Each network demand is identified by the tuple: source router, destination site, and traffic class. Traffic class is a differentiated service queue name *e.g.,* voice, interactive, best-effort, or scavenger [5]. Tier-0 traffic uses best-effort or higher traffic classes. Tier-1 and tier-2 use the scavenger traffic class. The data feeds of the demand predictor are independently scaled out and not part of the controller.

Traffic Engineering Scheduler forms the core of the BLASTSHIELD system (Fig. 3). It ingests global network topology and global demands from topology service and demand predictor respectively. The path computer calculates paths using the dynamic topology for the source-destination pairs in the global demands. MaxFlow path computer uses maximum flow algorithms [14], and penalizing path computer computes risk diverse shortest paths using Dijkstra. Path *constraints*, described later in §§ 5.1 and 5.2, limit allowed paths in order to support the routing in BLASTSHIELD.

TE solver consists of a chain of linear programming optimization steps that place demands on multiple paths with unequal weights between demand source and destination pairs. It places tier-0 demands on paths with diversity protection that minimize latency subject to approximate max-min fairness. Lower priority demands in tier-1 and tier-2 classes are placed on paths that minimize the maximum link utilization. For brevity, we exclude the optimization problem formulations, which are previously described in [6, 16, 21, 25].

The FIB generator mechanically converts the output of the TE solver, called the *solver result*, into TE routes. The *slice configuration* specifies the subset of routers for which routes are generated. The FIB generator transforms the solver result based on the slice configuration, and produces routes only for the routers in the slice. The network is re-optimized every 3 minutes, or on topology change, whichever occurs first.

Route Programmer programs traffic engineering routes in the router agent which in turn installs them in the router. It periodically receives the full set of routes for all slice routers from the traffic engineering scheduler. This is called the traffic engineering forwarding information base (TE FIB). The FIB is organized into per-router flow and group tables (see Fig. 4). The route programmer updates all slice router agents in parallel using an update procedure, called *make-before-break*. The principle is to make all new traffic engineered paths before placing traffic on them. Intermediate FIBs build new paths, transfer traffic to the new paths, and tear down unused paths.

Router Agent runs on all WAN routers. It installs TE routes, monitors the end-to-end liveness of TE paths (*tunnels*), and modifies ingress routes based on liveness information. Route installation on the router requires translating the FIB into router platform-specific API calls. Router agents have a platform-dependent module to handle this translation. The router agent verifies tunnels within the slice using probes generated natively or with BFD [22] from tunnel ingress points.

Flows are unequally hashed to live paths based on the path weight, flow 5-tuple, and traffic class. If a path goes down, the agent proportionally distributes the weight of the down path to remaining up paths. If no path is up, then the ingress route is withdrawn, and packets are forwarded using switch-native protocol routes. This is called *local repair*.

4.2 Design considerations

Global solution at local instances. Each BLASTSHIELD slice controller consumes global network topology and demands. The solver of each controller computes flow allocations for the entire network. Therefore, each slice controller produces the same solver result if its inputs and solver software versions are the same. In practice, inputs and software versions can differ, and we study the impact of these differences in § 6.2. Although a slice controller only programs the WAN routers in its slice, it optimizes flow with a global view. Slice controllers do not communicate with each other but gather inputs from the network. Performing global optimization at each slice controller is beneficial while deploying changes to the network. Some faults involve complex interactions that only occur in unique parts of the WAN. Global inputs increase the coverage of code paths while new software or configuration changes are being deployed in small blast radius slices.

Slices as isolated routing domains. In centralized TE systems, a single controller is responsible for programming all WAN routers with the TE routes. BLASTSHIELD replaces the centralized controller with multiple slice controllers that can only program the routers within their slice. By preventing slice controllers from programming routers outside their slice, we enforce fault isolation between slices. In addition, the routing mechanisms described in § 5 ensure that the failure of one controller does not impede other controllers *e.g.,* the failure of a downstream slice controller on an inter-slice route in the WAN does not lead to blackholing of traffic. Similarly, slice controllers with inconsistent views of the network, route packets to their destination without centralized control.

Fault tolerant design. All services run on multiple machines in at least two geographically separate clusters. Topology service instances are fully active, but elect a leader to avoid oscillations if two instances report different topologies due to faults or transients. The traffic engineering scheduler and route programmer elect leaders, and switchover in case of failure. The route programmer handles all the faults and inconsistencies that can happen during programming, *e.g.,* router agents are unresponsive or have faults before, during, or after route programming. Reliable controller-agent communication is achieved by using network control traffic class, and redundant data and management plane connections. The router agent can react to network faults even when it is disconnected from the router programmer.

Decoupling TE scalability from blast shielding. BLAST-SHIELD employs slice controllers to reduce the blast radius of faults in our network. We handle scale along several dimensions, unrelated to blast shielding. But slices also provide the following scaling benefits. The total number of tunnels in the network decreases because an inter-slice path is a sequence of intra-slice tunnels in BLASTSHIELD, whereas in SWAN it required its own tunnel. Second, shorter tunnels decrease tunnel probe round-trip times and speed up local repair.

5 Routing and forwarding in BLASTSHIELD

The routing of *intra-slice* flows in BLASTSHIELD is the same as SWAN. In this section, we describe BLASTSHIELD's extensions to enable routing and forwarding of *inter-slice* flows *i.e.,* flows whose traffic engineered paths span multiple slices. § 5.1 describes inter-slice routing, the approach we deployed, and § 5.2 describes a source routing approach that was evaluated but not deployed.

5.1 Inter-slice routing

In SWAN, packets are routed using a combination of switch-native protocols and the TE controller. WAN routers connected to the datacenter fabric advertise datacenter routes with themselves as the BGP [27] next hop. BGP receivers recursively lookup the route for this BGP next hop and find multiple available routes: the shortest path route computed by the IGP, or the route programmed by the TE controller which leverages traffic engineered paths. TE routes have higher precedence than the IGP routes. The TE route encapsulates packets using Multiprotocol Label Switching (MPLS) [28] path labels from a label range reserved for the TE controller.

BLASTSHIELD routes inter-slice flows *i.e.,* flows whose traffic engineered paths span multiple slices, using *slice-local encapsulation* till the slice boundary. Slice controllers add encapsulation headers while the packet is within the slice but ensure that the packets arrive at the next slice in their *native encapsulation i.e.,* the encapsulation in which the packets entered the WAN. Each slice controller is only responsible for routing traffic to the ingress router of the next slice. Packets are encapsulated with an MPLS path label at the time of BGP route lookup on the WAN ingress router or the intermediate slice ingress routers. In both scenarios, transit routers forward the packet using the MPLS path label, and the label is popped by the penultimate router — either at a slice boundary or at the destination. Intra-slice traffic is split across TE paths only once at the WAN ingress router. Inter-slice traffic can also be split at the ingress router of an intermediate slice.

Inter-slice forwarding In Fig. 4, all four slice controllers determine that the demand from *a* to *z* should be placed on paths *abegjuwxz*, *acdmoqstyz*, and *acdmonikvyz* with weights 0.3, 0.42, and 0.28 respectively. Slice 1 programs *abe* with weight

Slice 1 routes to z				
Device	Prefix	Wt	Action	Out
a	z	0.3	push 151	ab
		0.7	push 157	ac
b	151	-	pop	be
c	157	-	swap 157	cd
d	157	-	pop	dm

Slice 4 routes to z				
Device	Prefix	Wt	Action	Out
u	z	1	push 443	uw
v	z	1	push 447	vy
y	z	1	-	yz
w	443	-	swap 443	wx
x	443	-	pop	xz
y	447	-	pop	yz

Slice 2 routes to z				
Device	Prefix	Wt	Action	Out
e	z	1	push 223	eg
i	z	1	push 227	ik
g	223	-	swap 223	gj
j	223	-	pop	ju
k	227	-	pop	kv

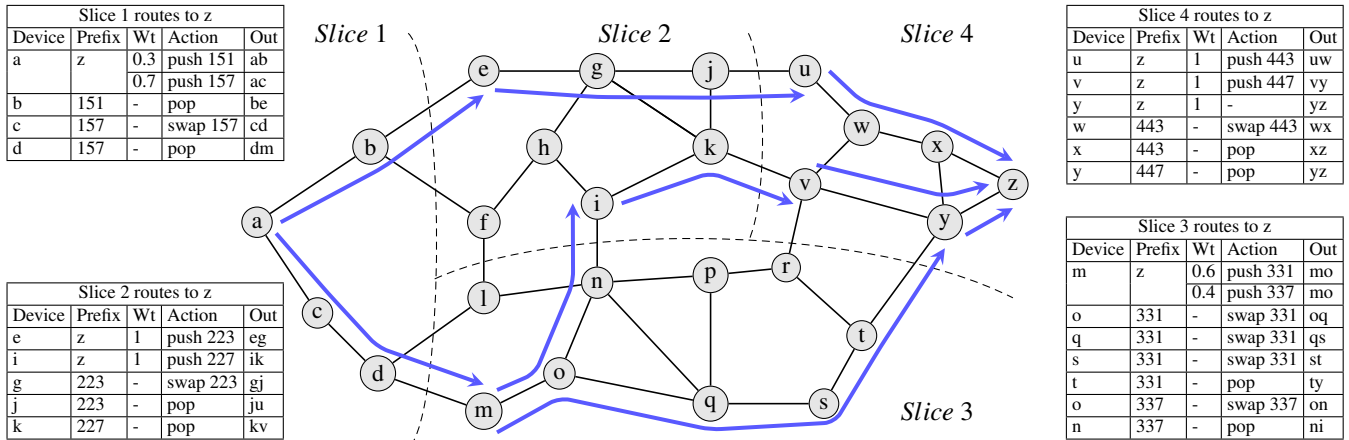| Slice 3 routes to z				
Device	Prefix	Wt	Action	Out
m	z	0.6	push 331	mo
		0.4	push 337	mo
o	331	-	swap 331	oq
q	331	-	swap 331	qs
s	331	-	swap 331	st
t	331	-	pop	ty
o	337	-	swap 337	on
n	337	-	pop	ni

Figure 4: Inter-slice routing using an example router-level network graph divided into four slices. The tables represent TE FIBs programmed by slice controllers using inter-slice routing. Each slice controller programs the path segment within its slice. For the path *abeg juwxz*, slice 1 programs *abe*, slice 2 programs *eg ju*, and slice 3 programs *uwxz*. Traffic arriving at slice ingress routers get encapsulated and split over different paths. Transit routers guide the packet along the path specified by the MPLS label. Packets return to native encapsulation at the next slice and the WAN exit.

0.3, and *acdm* with weight 0.7. Slice 2 programs *eg ju* and *ikv*. Slice 3 programs *moqsty* with weight 0.6, and *moni* with weight 0.4, and slice 4 programs *uwxz*, *vyz*, and *yz*. Controllers only need to install routes in their slice routers.

If any downstream slice controller fails to program routes to the destination, packets are forwarded using protocol routes along the shortest paths to the destination. Since we enable segment routing [11] with the IGP, the IGP route changes the packet encapsulation and routes the packet to the destination. For example, if the slice 2 controller withdraws all routes due to a failure, the inter-slice traffic uses shortest paths to the destination, *z*. This is the blast ripple of a down controller. In § 6.1, we will discuss how to define slice boundaries to decrease the blast ripple. Downstream slice controllers may have slightly inconsistent views due to network events like link flaps. Inter-slice traffic will be forwarded on shortest paths while the controllers converge. We show results on the alignment of multiple controllers in § 6.2.

Preventing routing loops. Unlike the TE controller in SWAN, a BLASTSHIELD slice controller is only responsible for routing packets within the slice and not until the packets' destination. Since each slice is its own routing domain, inconsistent views of the global network graph in different slice controllers can lead to routing loops.

BLASTSHIELD avoids routing loops by enforcing *enter-leave constraints* on inter-slice next hops. These constraints define the set of inter-slice next hops for all source-destination pairs in the network. The constraints ensure loop-free paths and are calculated offline using a static network graph. The path computer calculates paths on the dynamic network graph, and only allow paths that satisfy the enter-leave constraints. However, enter-leave constraints should not be overly restrictive. For example, a potential approach to preventing routing loops can limit inter-slice next hops to be on the minimum

spanning tree from the source router to the destination. But this approach restricts inter-slice paths to go through a few links and causes bottlenecks.
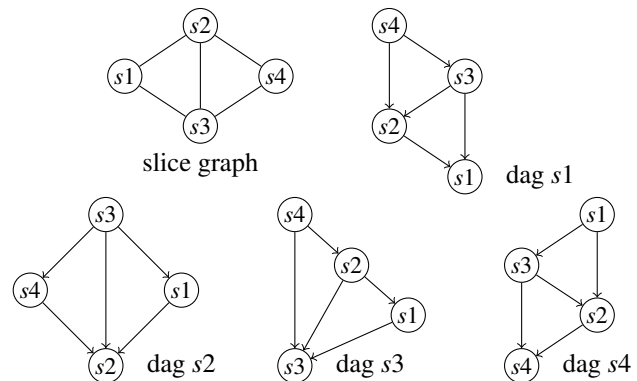

Figure 5: Enter-leave constraints restrict paths to achieve loop-free routing. Slice graph is a component level graph of Fig. 4. Slice DAGs are constructed from shortest path distances in the slice graph. Router-level paths must follow DAG edges when crossing slice boundaries. Path *acdmonikvyz* is allowed for TE because $s1 \rightarrow s3 \rightarrow s2 \rightarrow s4$ is a path in DAG *s4*. Path *abf hinprvyz* is not allowed for TE because $s2 \rightarrow s3$ is not present in DAG *s4*.

Computing enter-leave constraints. An offline generator computes enter-leave constraints from the static router-level network graph to prevent inter-slice routing loops. It first constructs a slice graph from the network graph, where each slice node represents a strongly connected component (SCC) after removing all inter-slice links. Figure 5 is the slice graph of Fig. 4, formed by removing inter-slice links *be*, *bf*, *dl*, *dm*, *fl*, *in*, *ju*, *kv*, *rv*, and *ty*, and calculating SCCs. A slice can contribute one or more SCCs as nodes to the slice graph. A link between the slice graph nodes aggregates all links between SCCs in the network graph. Link weights in the slice graph are computed from link weights in the network graph.

The enter-leave constraint generator then constructs per-destination slice DAGs based on the shortest path distances in the slice graph. The enter-leave constraints come out directly from the slice DAGs. In Fig. 5, the slice DAG for $s4$ says that paths from any node in $s1$ to any node in $s4$ can only have inter-slice transitions: $s1 \rightarrow s2 \rightarrow s4$, $s1 \rightarrow s3 \rightarrow s4$, and $s1 \rightarrow s3 \rightarrow s2 \rightarrow s4$. No controller, no matter its topology, can use any other inter-slice transition.

The path computer blacklists edges excluded by enter-leave constraints in the dynamic network graph before computing TE paths. Since the slice DAG is loop-free, paths computed by any slice controller are also loop-free. This ensures that even if slice controllers have inconsistent views of the dynamic network graph, they will arrive at loop free routes. Enter-leave constraints place restrictions on TE paths, and reduce the number of paths available to place demands. We evaluate the percentage of allowed paths vs. computed paths without constraints in § 6.1.

Verifying enter-leave constraints. Due to the negative impact of routing loops in production, and because they are global configuration, enter-leave constraints are verified offline before deployment. Enter-leave constraints are updated when there are newly provisioned routers or inter-slice links in the network. They do not need to be updated for newly provisioned intra-slice links.

We use the following formalism to define correct inter-slice routing. Let $\mathcal{R}$ be the set of defined route keys, where route key is a tuple of (router, destination prefix), **end** be the terminating route key, **null** be the undefined route key, and ttl be the packet time to live. Let $f : \mathcal{R} \rightarrow \mathcal{R}$, where $f(\textbf{null}) = \textbf{null}$, $f(\textbf{end}) = \textbf{end}$. Routing is a repeated application of $f()$, till $f^n(x) = \textbf{end}$ where n ranges over $1 \leq n \leq ttl$. The collection of TE, BGP, and the IGP routes, and their union are examples of routing functions. The routing function is complete, loops, or blackholes, if:

$$\forall x, \exists n : \quad f^n(x) \ = \textbf{end} \quad \text{(complete)}$$
$$\exists x, n : \quad f^n(x) \ = x \quad \text{(routing loop)}$$
$$\exists x, n : \quad f^n(x) \ = \textbf{null} \quad \text{(blackhole)}$$

where x ranges over $\mathcal{R} \setminus \{\textbf{end}, \textbf{null}\}$ and n ranges over $[1..ttl]$. Enter-leave constraints are verified using this formalism to detect routing loops.

5.2 Why not source routing?

In this section, we describe an alternate approach that leverages the capabilities of segment routing (SR) [11], and why we did not adopt this approach.

Loose source routing with SR. SR is a source-based routing technique that allows senders to specify the packets' route through the network by leveraging the MPLS forwarding plane. An SR router subjects arriving packets to a policy and encapsulates the matching packets in an MPLS label stack, each label represents a *segment* in the SR-path. A *node segment* causes the packet to be routed on least-cost paths computed

by the IGP to the router identified by the node segment. An *adjacency segment* causes the packet to use a specified link for its next hop.

An IGP path computer models the modified Dijkstra shortest path first algorithm [18]. Coupled with segment identifiers from topology service (§ 4.1), it implements *loose source routing*. In place of explicitly listing adjacency segments of hop-by-hop links of a path, loose source routing uses a node segment when it exactly represents the sequence of the hop-by-hop links of the path. Figure 6 shows an example of loose source routing for the same paths shown in Fig. 4. The path *begjuwxz* is composed of two shortest path segments *begju* and *uwxz*. Hence a encapsulates with label stack of $[n(\text{u}) \, n(\text{z})]$ to route to z, where $n()$ is the node segment identifier of a router.

Packet encapsulations reduce hashing entropy. To achieve balanced utilizations across links in the WAN, the cloud network employs two load balancing mechanisms. Link aggregation group hashing sprays packets on member links of a port-channel. Equal cost multi-path hashing sprays packets on the next hops of a group of traffic engineering routes. The packet processor uses fields from the packet headers to hash the packet to different output ports with the goal of maximizing entropy in the hash calculation. To achieve high entropy, the outermost IPv4/IPv6 source and destination addresses under stack of MPLS header encapsulations should be used to calculate the hash. A deep MPLS label stack can impair the ability of the packet processor to extract the relevant fields in the IP header.

The *depth limit* is the maximum number of MPLS encapsulations a packet can have while still allowing the packet processor to extract the header fields of the original (*i.e.,* prior to MPLS encapsulations) packet. The depth limit is switch platform-dependent [2, 8, 20]. We note that if the packets entering the WAN are already encapsulated in MPLS, the depth limit available to source routing is further reduced.

Why select inter-slice routing? Based on the current generation of platforms across different regions of our cloud WAN, the depth limit is four labels. Paths that require more labels cannot be used for TE. Figure 7 studies the label stack depth needed to encode paths computed by the path computer for current and future evolutions of the WAN. In source routing, 45% of computed paths can be used for TE. For comparison, 69% of computed paths can be used for TE in inter-slice routing (see § 6.1).

Second, in source routing, a downstream slice can only transit upstream flows. In inter-slice routing, the downstream slice is free to rebalance the traffic to correct errors made upstream or mitigate for local slice conditions. This kind of control is not available with source routing.

Source routes to z				
Device	Prefix	Wt	Action	Out
a	z	0.3	push $n(u)$ $n(z)$	ab
		0.42	push $n(o)$ $n(z)$	ac
		0.28	push $n(o)$ $n(k)$ $n(z)$	ac

Protocol routes for *cdmoqstyz*			
Device	Prefix	Action	Out
c	$n(o)$	swap $n(o)$	cd
d	$n(o)$	swap $n(o)$	dm
m	$n(o)$	pop	mo
o	$n(z)$	swap $n(z)$	oq
q	$n(z)$	swap $n(z)$	qs
s	$n(z)$	swap $n(z)$	st
t	$n(z)$	swap $n(z)$	ty
y	$n(z)$	pop	yz

Protocol routes for *beg juwxz*			
Device	Prefix	Action	Out
b	$n(u)$	swap $n(u)$	be
e	$n(u)$	swap $n(u)$	eg
g	$n(u)$	swap $n(u)$	gj
j	$n(u)$	pop	ju
u	$n(z)$	swap $n(z)$	uw
w	$n(z)$	swap $n(z)$	wx
x	$n(z)$	pop	xz

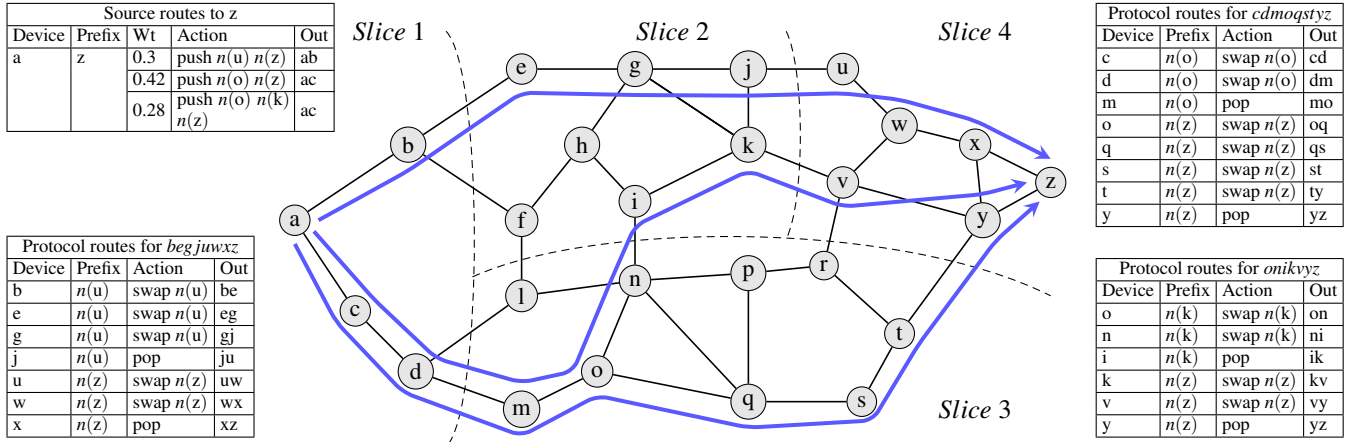| Protocol routes for *onikvyz*			
Device	Prefix	Action	Out
o	$n(k)$	swap $n(k)$	on
n	$n(k)$	swap $n(k)$	ni
i	$n(k)$	pop	ik
k	$n(z)$	swap $n(z)$	kv
v	$n(z)$	swap $n(z)$	vy
y	$n(z)$	pop	yz

Figure 6: Source routing. Slice 1 controller programs ingress routes to z using loose source routing. The IGP with segment routing takes care of transit routes. The path *beg juwxz* is composed of two shortest path segments *beg ju* and *uwxz*. Hence the label stack for the path is $n(u)$ $n(z)$, where $n()$ is the node segment identifier of a router. Weights of intra-slice links are 1 and inter-slice links are 5.
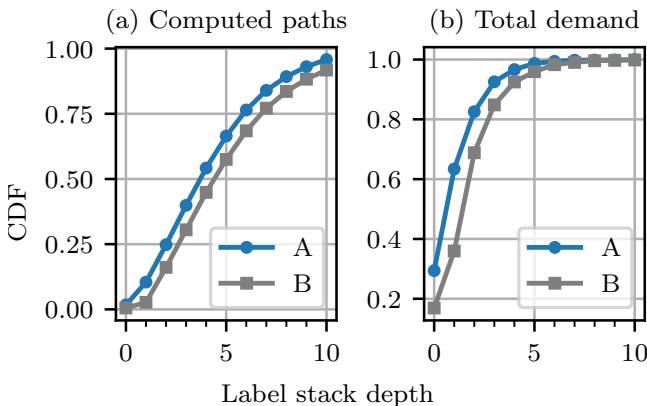

Figure 7: Cumulative distribution function of (a) computed paths and (b) total demand, by label stack depth for inputs A and B of increasing sizes. If depth limit is four, 45% of computed paths and 93% of the total demand map to allowed paths for input B.
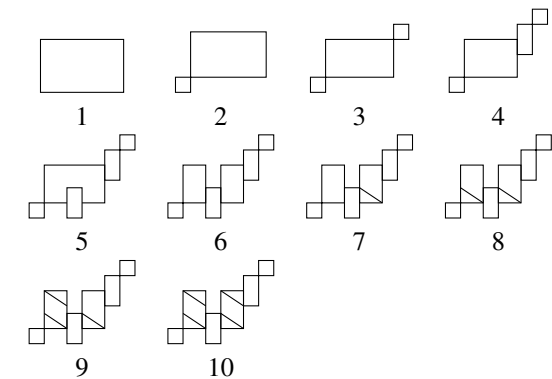

Table 2: Ten slice configurations of the global cloud network. In (1) the entire network is one slice. Slices 2–6 are formed by grouping routers in geographies. Slices 7–10 are created by further subdividing the two largest geographies, Europe and North America.

6 Evaluating BLASTSHIELD in production

The incremental deployment of BLASTSHIELD began in 2020 and today BLASTSHIELD has replaced the legacy SWAN traffic engineering system in Microsoft's cloud network. In § 6.1, we evaluate the benefits and costs of WAN slicing using demands and topology inputs from the Microsoft backbone network for the month of July 2021. The benefit of slice-decentralized traffic engineering is the reduction in traffic loss from a slice failure. Its cost is the reduction in TE throughput due to enter-leave constraints. We quantify cost and benefit as we incrementally divide the global network into ten slices. In § 6.2, we evaluate the stochastic effects caused by multiple and independent BLASTSHIELD controllers. We show that despite the controllers having different configurations, software versions and network topology snapshots, they arrive at nearly similar flow allocations.

6.1 Availability vs. throughput trade-offs

We incrementally carve out slices from the global cloud network as shown in Table 2. We consider ten different slicing configurations with increasing number of slices from 1 to 10. Slice configuration 1 represents centralized traffic engineering as in SWAN. Slice configurations 2–6 are formed by drawing slice boundaries around large geographical regions like APAC, EMEA, India, North America, Oceania, and South America. In Table 2, slice configuration 2 represents the network divided into two slices: India and the rest of the world, configuration 3 represents India, Oceania, and the rest of the world, and so on. Slices 7–10 are formed by additionally dividing the two largest geographies, Europe and North America, into smaller slices. In our network, configurations 1–6 tend to have higher intra-slice traffic in comparison to inter-slice traffic. Slices have up to three strongly connected components, arising from disconnected sites and router planes.

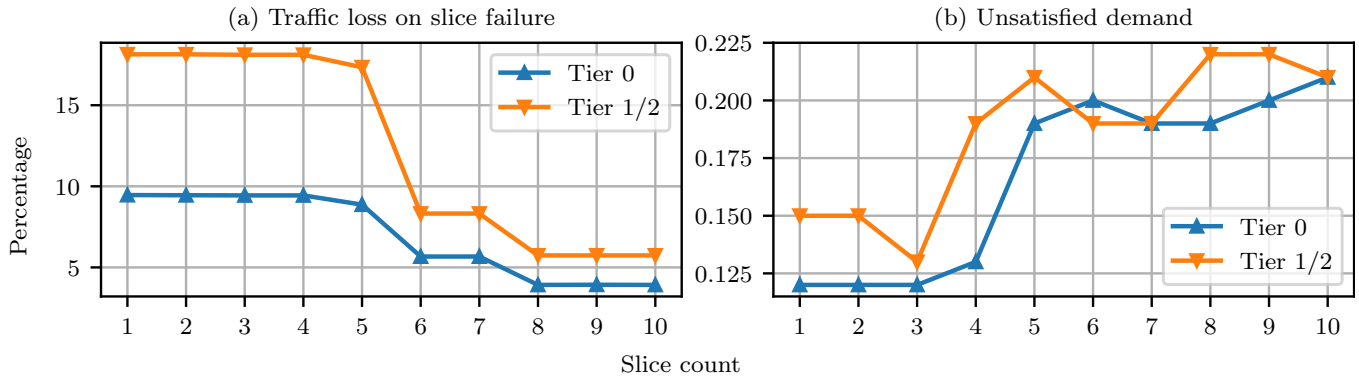**Availability gains from decentralized TE.** The key benefit

Figure 8: Benefit of BLASTSHIELD compared to its cost as a function of slice count: (a) Traffic loss from worst case single slice failure as a percentage of requested demand, (b) Unsatisfied demand due to enter-leave constraints as a percentage of requested demand.
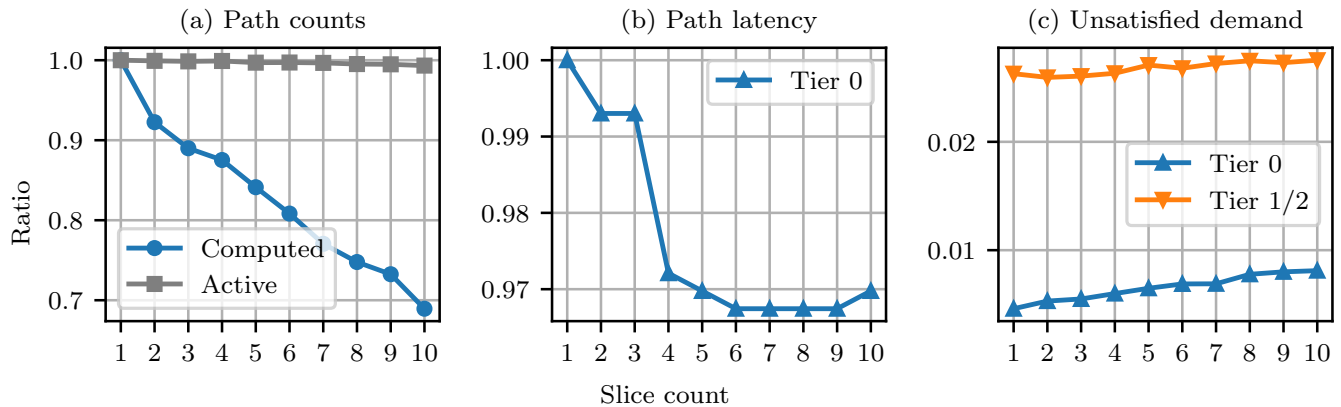

Figure 9: Stress-testing BLASTSHIELD with worst-case failures and 2× demands. (a) Computed paths are the count of paths computed with enter-leave constraints. Active paths are the count of paths used for traffic engineering. (b) Path latency is the traffic weighted average latency of active paths for tier-0 demands. (c) Unsatisfied demand is the unallocated demand per traffic tier. All values, except unsatisfied demand, are normalized to corresponding values for one slice; the latter is a ratio of unsatisfied to requested demand.

of BLASTSHIELD's slicing is the reduction in blast radius when a slice controller fails. We consider the failure where the slice controller removes all programmed TE routes. This causes the traffic to fall back on protocol routes and the ensuing traffic loss is the impact of the slice failure. We measure the traffic loss using a network simulator because the scenarios we are testing cannot be replicated in production. The inputs to the simulation are the production network demands, topology, TE and the IGP routes, and the network simulator models routing, forwarding, and queuing behavior. The simulator is used internally for capacity planning and operational safety checks, and hence is a well-tested proxy for the production network.

Figure 8 (a) shows the impact of the worst-case single slice failure when BLASTSHIELD is operating with 1–10 slices. We keep the demands and topology fixed in this experiment. For each slice configuration, we fail the largest slice by demand. The network uses the IGP routes of the failed slice and TE routes of the remaining slices (if any) to allocate the remaining demands. The traffic losses are caused by congestion due to shortest path routing over IGP routes. There are no losses due to traffic blackholes or routing loops. Figure 8 (a) shows that with ten slices, tier-0 traffic loss due to slice failure, which is

the metric for blast radius, decreases by 60%, from 9.5% to 3.9%. Tier-1 and tier-2 traffic loss reduction is greater at 70% (18.1% to 5.7%) because they map to scavenger traffic class and experience more congestion losses when the failed slice uses IGP routes. Slices 2–4 show little improvement because the largest slice can still cause an overly large failure. The improvements come at six and eight slices with the breakup of Europe and North America into separate and smaller slices.

Throughput cost of decentralized TE. The key reason why inter-slice routing in BLASTSHIELD can have lower throughput than SWAN is due to the enter-leave constraints (§ 5.1). These constraints decrease the choice of paths available for placing demands, which in turn decreases the demands that can be allocated. Figure 8 (b) shows unsatisfied demand from enter-leave constraints as a percentage of requested demand. We calculate worst case unallocated demand from 20 variations of the network topology, each variation has multiple shared risk failures that reduce the available capacity. Without constraints, the worst-case unsatisfied demand is 0.27% of the requested demand, and with ten slices it increases to 0.42%. The increase in unsatisfied demand of 0.15% is much smaller than the 18% traffic loss reduction from slice failure. Addi-

tional capacity can be provisioned to decrease the unsatisfied demand.

Stress testing BLASTSHIELD. We oversubscribe the network by doubling the bandwidth values in requested demands, and test with variations of the production network with multiple shared risk group failures in hot spots of the topology. The purpose of the stress test is to evaluate the performance of enter-leave constraints in the presence of significant oversubscription. Figure 9 shows the impact of slicing on paths computed by the BLASTSHIELD path computer. Since the constraints enforce a shortest path order when crossing slice boundaries, they exclude paths that would otherwise be allowed. At ten slices, computed paths decrease by 31% when compared with one slice. The number of paths actively used for carrying traffic decreases slightly — by $< 1\%$ due to some demands remaining completely unsatisfied, or diverse paths not getting found. Figure 9 shows that the traffic weighted path latency of tier-0 demands decreases by 3% because the computed paths are skewed towards shortest paths. Finally, unsatisfied demands as a percentage of requested demands increases 16% from 3.1% to 3.6%. Unsatisfied demand increases at half the rate of computed path decrease which is well controlled. In practice, the percentage of computed paths allowed by enter-leave constraints are used to determine whether a slice strategy is appropriate.

6.2 Stochastic effects of multiple controllers

Prior to the deployment of BLASTSHIELD, the centralized SWAN controller programmed new TE routes for the entire cloud network. BLASTSHIELD replaces the centralized controller with multiple slice controllers that snapshot the network topology and demands at different times. Moreover, the controllers may re-run the TE optimization and program their slice routers at different times. We study the impact of the temporally staggered operation of slice controllers to ask: can multiple slice controllers work harmoniously and not be discordant?

We reserve 15% scratch capacity in order to support the high SLO of tier-0 traffic. Transient traffic bursts and hashing polarization can cause link utilization to differ significantly from expected values. The scratch capacity is used to avoid congestion losses in these conditions. BLASTSHIELD uses this scratch capacity to deal with differences that arise with multiple controllers.

Symphony or cacophony of controllers? Path weights decide the split of traffic across paths and are the ultimate result of TE optimization. The weight of a path is the fraction of demand placed on it. BLASTSHIELD programs the newly computed path weights every 3 minutes. Since all slice controllers solve the TE problem for the entire network, we measure if the path weights that different controllers compute diverge from each other. We quantify the *path weight difference* as

the root mean squared error between path weight time series from two controllers. A path weight difference of zero implies that the controllers are perfectly aligned. Non-zero path weight difference implies that the controllers are setting aside different link bandwidths for a flow which can cause congestion.

We measure the path weight difference between six different BLASTSHIELD controller pairs in the production network over a 30-day period. There were days when the controllers were operating with different configurations, different software versions, in addition to network topology and demand changes that happen throughput the day. Figure 10 shows that only 2% of paths and 3% of total demand have path weight difference of ≥ 0.15. Inter-slice demands make up 48% of paths but 10% of total demand because of the slicing strategy. Since intra-slice traffic dominates, the impact of the path weight difference is limited. The slicing strategy and scratch capacity allow multiple controllers to operate without coordination.

Solver stability. Different path weights for *slightly perturbed* inputs can create an operational challenge for BLASTSHIELD. We constrain the solver models to make their solutions stable — the tier-0 objective function minimizes demand weighted latency after solving for max-min fairness. In practice, this makes the solver results more stable when subjected to input perturbations. We do not allow non-determinism in the TE solver *e.g.,* no parallel primal and dual simplex invocation in the linear programming solver to pick the first result, since they will produce different solution vectors that result in different path weights.

We evaluate the stability of the solver results using the normalized autocorrelation function (ACF) $\rho(\tau)$. ACF is the correlation of a time series to a delayed version of itself, as a function of the delay, τ. In Fig. 11, we calculate ACF for the hour-long path weight time series of all paths in the production network over a 24-hour period. ACF values range $[-1, 1]$, and 1 implies perfect correlation.

Demand and network topology changes also affect path weight ACF. So perfect correlation is not possible in an operational network. Figure 11 (a) is an example path weight time series with ACF(30 minutes) of 0.65 showing steady values of the same path weight interspersed by occasional gyrations. Figure 11 (b) shows that mean ACF is 0.75–0.63 for lags of 3–30 minutes. This reaffirms the data in Fig. 10 that path weights from independent BLASTSHIELD controllers are predominantly the same.

7 Discussion

In this section, we discuss our operational experience with BLASTSHIELD and describe safe deployment of software and configuration in BLASTSHIELD slices. We consider the implications of byzantine slice controllers, and the safeguards in place to prevent damage from them.
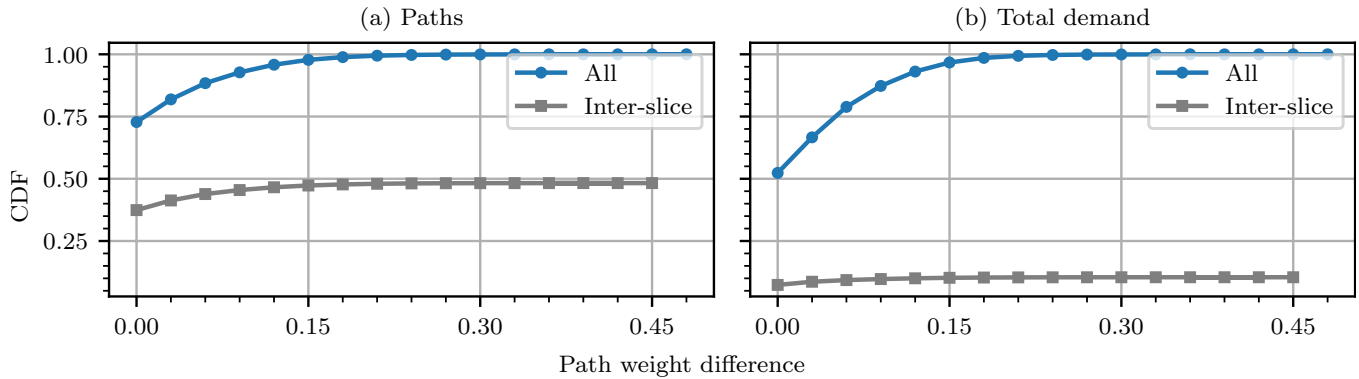
Figure 10: Cumulative distribution function of (a) paths and (b) total demand, by path weight difference, for all demands and inter-slice demands, measured for six controller pairs in the production network over a 30-day period. 98% of paths and 97% of total demand have path weight difference ≤ 0.15. Inter-slice demands make up 48% of paths but 10% of total demand.
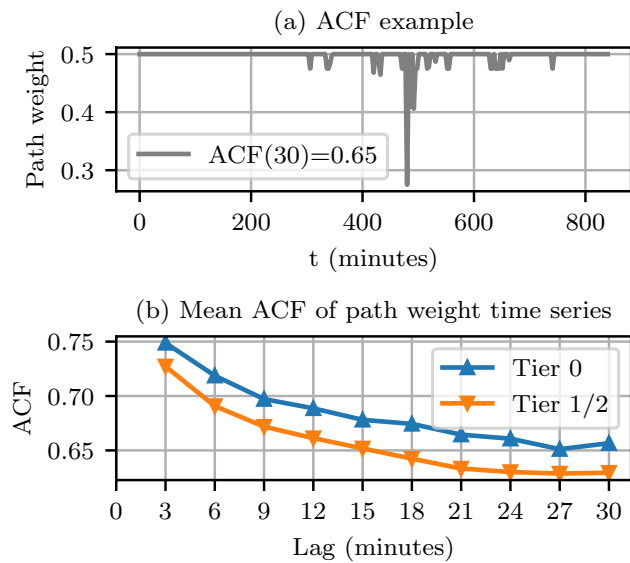

Figure 11: Autocorrelation function (ACF) measures self-similarity with a delayed version, and ranges $[-1, 1]$ with 1 being perfect correlation. (a) Example path weight time series with ACF(30 minutes) of 0.65. (b) Mean ACF of path weight time series averaged over all paths in the production network over a 24-hour period by traffic tier.

7.1 Operational experience

BLASTSHIELD has been in operation for two years. Migration from SWAN to BLASTSHIELD was carried out over a number of months. The first step was to deploy inter-slice routing and forwarding functionality in the SWAN controller and router agents. This was the riskiest step and preceded by many months of testing in a virtualized emulation environment of the production network with fault injection. Each slice migration involved preparing a new BLASTSHIELD controller, excluding a set of routers from the slice configuration of the SWAN controller, and adding them to the new controller.

To support deployment of new software and configuration changes, we define slices that range from low to high impact. *Safe deployment* is a partial ordering of slices based on their blast radius. BLASTSHIELD has two staging sites with a staging controller, and new software and configuration is first deployed here. The next slice has the smallest production scope. We assign routers in geo-redundant site pairs to separate slices for additional safety. Deployment progresses to the next slice in the sort order after a sufficient probationary period. The process continues till either all slices receive the new version of software or a failure happens in a slice, which may trigger a rollback of this version from all slices.

Enter-leave constraints have been updated multiple times to pick up newly provisioned routers and links. In one instance, the constraints affected traffic engineering for an inter-datacenter pair by excluding too many links. New constraint configuration to correct the error and reverse an inter-slice traffic flow was deployed without incident.

We have introduced new hardware platforms, router agents, and controller services that would be considered high risk in the SWAN paradigm. BLASTSHIELD allowed us to introduce new implementations in isolated slices with very small blast radius and no inter-slice traffic. Initially the slice only served intra-slice traffic. Inter-slice traffic was introduced after the slice had been in operation for many months. Outages caused by failures in the new slices never had a global impact.

Slice controller environments are used by additional services to decrease their blast radius. For example, discretionary flows can be throttled at the sending host to control congestion in the network. Bandwidth is allocated to discretionary flows by global optimization but each controller only serves bandwidth pools for a smaller fault domain.

7.2 Byzantine slice controllers

A byzantine controller is an unreliable controller that is disseminating false information or sabotaging the operation of other slices in the network [24]. A controller that only impacts its own traffic is not considered byzantine in our analysis.

Resistance to byzantine slice controllers is baked into the BLASTSHIELD design. BLASTSHIELD does not allow any inter-controller interaction. Each controller uses its own services to get demand and topology inputs. It calculates TE

routes by sensing the state of the network, and does not rely on communication with other controllers. Route programmers of a WAN slice do not communicate with router agents in other slices, and thus are unaffected by unreliable agents in other slices. Access control lists on slice routers prevent another slice controller from attempting to program them.

Despite these protections, a byzantine controller may route traffic in a way that causes congestion in downstream slices. A slice controller estimates the demands at the slice boundary based on the assumption that all slices are well behaved *i.e.,* they use the same algorithm and configuration as itself. Byzantine slice can violate this assumption. The impact of a byzantine controller's actions are limited to the remote traffic from the byzantine slice. WAN traffic patterns inform the creation of slices that minimize inter-slice traffic [30].

We note that non-byzantine controller faults are also possible. Faulty controller may withdraw all its routes and congest links in its own or other slices. A faulty controller may loop or blackhole packets. While we have safety checks and routing constraints that prevent such conditions, if a controller manages to bypass the checks, human intervention is required. We mitigate these failures by pausing the faulty controllers, and restoring the network programming to last known good FIB.

8 Related work

B4 [17, 19] and EBB [10] are two examples of operational networks that use software-defined traffic engineering. [17] states that site-level domain controllers were large blast radius and faults caused widespread impact to traffic passing through the affected site, which led them to divide a site into two or four control domains, each managed by a separate domain controller. Similarly, in BLASTSHIELD, we assign routers in a site to separate slice controllers. [17] uses a central controller to calculate tunnel split groups and the sequencing of traffic engineering operations, and a large fleet of domain controllers to do route calculation and programming. BLAST-SHIELD does not use any central controllers and each slice controller performs global traffic engineering calculation and slice-local route programming. It should be noted that the network architectures of BLASTSHIELD and B4 are quite different. [10] uses a centralized controller and segment routing, which we evaluated but did not select because of label stack depth and lack of control in intermediate slices.

Prior work on software-defined traffic engineering [1, 7, 23, 25, 29] focus on the optimization problem of maximizing utilization, guarantee fairness, preventing congestion under faults, or dynamic pricing without considering how they would be deployed. They all assume a centralized controller will perform the optimization for the entire network without considering what happens when the controller fails. BLAST-SHIELD can be used in conjunction with these works to make them deployable in operational networks.

Inter-slice routing is similar to pathlet routing [13] but without any controller interaction or dissemination protocol. [9, 12] study consistent updates and loop avoidance with a centralized controller, but not multiple controllers with inconsistent views. BLASTSHIELD adopts a stricter approach of not communicating with another controller to avoid additional failure modes from faults in the communication, and because the information a controller needs can be acquired from the network.

9 Conclusion

In this work, we motivate the design of a decentralized traffic engineering system for large-scale cloud WANs using our operational experience with SWAN. We propose BLASTSHIELD, Microsoft's new global TE system that decentralizes the TE controller with WAN slicing and implements loop-free inter-slice routing. BLASTSHIELD achieves similar throughput as fully centralized TE implementations while significantly reducing the blast radius of faults in TE controllers. We have been operating Microsoft's WAN with BLASTSHIELD, and it has substantially lowered the risk of configuration changes causing large outages.

References

[1] Firas Abuzaid, Srikanth Kandula, Behnaz Arzani, Ishai Menache, Matei Zaharia, and Peter Bailis. Contracting wide-area network topologies to solve flow problems quickly. In *Proceedings of USENIX NSDI*, pages 175–200, April 2021.

[2] Port channels and LACP load balancing hashing algorithms. https://www.arista.com/en/um-eos/eos-port-channels-and-lacp, accessed February 2022.

[3] Algirdas Avižienis. Fault-tolerant systems. *IEEE Transactions on Computers*, 25(12):1304–1312, December 1976.

[4] Daniel O. Awduche, Lou Berger, Der-Hwa Gan, Tony Li, Vijay Srinivasan, and George Swallow. RSVP-TE: Extensions to RSVP for LSP tunnels. RFC 3209, December 2001.

[5] Steven Blake, David L. Black, Mark A. Carlson, Elwyn Davies, Zheng Wang, and Walter Weiss. An architecture for differentiated services. RFC 2475, December 1998.

[6] Jeremy Bogle, Nikhil Bhatia, Manya Ghobadi, Ishai Menache, Nikolaj Bjørner, Asaf Valadarsky, and Michael Schapira. TEAVAR: striking the right utilization-availability balance in WAN traffic engineering. In *Proceedings of ACM SIGCOMM*, pages 29–43, August 2019.

[7] Yiyang Chang, Chuan Jiang, Ashish Chandra, Sanjay Rao, and Mohit Tawarmalani. Lancet: Better network resilience by designing for pruned failure sets. *Proc. ACM Meas. Anal. Comput. Syst.*, 3(3), December 2019.

[8] Implementing Cisco Express Forwarding. https://www.cisco.com/c/en/us/td/docs/iosxr/ncs5500/ip-addresses/66x/b-ip-addresses-cg-ncs5500-66x/m-implementing-cisco-express-forwarding-ncs5500.html, accessed February 2022.

[9] Szymon Dudycz, Arne Ludwig, and Stefan Schmid. Can't touch this: Consistent network updates for multiple policies. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 133–143, June 2016.

[10] Mikel Jimenez Fernandez and Henry Kwok. Building express backbone: Facebook's new long-haul network, May 2017. https://engineering.fb.com/2017/05/01/data-center-engineering/building-express-backbone-facebook-s-new-long-haul-network/.

[11] Clarence Filsfils, Stefano Previdi, Les Ginsberg, Brune Decraene, Stephane Litkowski, and Rob Shakir. Segment routing architecture. RFC 8402, July 2018.

[12] Klaus-Tycho Forster, Ratul Mahajan, and Roger Wattenhofer. Consistent updates in software defined networks: On dependencies, loop freedom, and blackholes. In *IFIP Networking*, May 2016.

[13] P. Brighten Godfrey, Igor Ganichev, Scott Shenker, and Ion Stoica. Pathlet routing. In *Proceedings of ACM SIGCOMM*, pages 111–122, August 2009.

[14] Andrew V. Goldberg, Éva Tardos, and Robert E. Tarjan. Network flow algorithms. In Bernhard Korte, Lásló Lovász, Hans Jürgen Prömel, and Alexander Schrijver, editors, *Paths, Flows, and VLSI Layout (Algorithms and Combinatorics)*, volume 9, pages 101–164. Springer-Verlag, 1990.

[15] Hannes Gredler, Jan Medved, Stefano Previdi, Adrian Farrel, and Saikat Ray. North-bound distribution of link-state and traffic engineering (TE) information using BGP. RFC 7752, March 2016.

[16] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven WAN. In *Proceedings of ACM SIGCOMM*, pages 15–26, August 2013.

[17] Chi-Yao Hong, Subhasree Mandal, Mohammad Al-Fares, Min Zhu, Richard Alimi, Kondapa Naidu B., Chandan Bhagat, Sourabh Jain, Jay Kaimal, Shiyu Liang, Kirill Mendelev, Steve Padgett, Faro Rabe, Saikat Ray, Malveeka Tewari, Matt Tierney, Monika Zahn, Jonathan Zolla, Joon Ong, and Amin Vahdat. B4 and after: Managing hierarchy, partitioning, and asymmetry for availability and scale in Google's software-defined WAN. In *Proceedings of ACM SIGCOMM*, pages 74–87, August 2018.

[18] Intermediate System to Intermediate System intra-domain routeing information exchange protocol for use in conjunction with the protocol for providing the connectionless-mode network service (ISO 8473). ISO/IEC 10589:2002, November 2002. https://www.iso.org/standard/30932.html.

[19] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jonathan Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined WAN. In *Proceedings of ACM SIGCOMM*, pages 3–14, August 2013.

[20] Understanding the algorithm used to load balance traffic on MX series routers. https://www.juniper.net/documentation/us/en/software/junos/sampling-forwarding-monitoring/topics/concept/hash-computation-mpcs-understanding.html, accessed February 2022.

[21] Srikanth Kandula, Dina Katabi, Bruce Davie, and Anna Charny. Walking the tightrope: Responsive yet stable traffic engineering. In *Proceedings of ACM SIGCOMM*, pages 253–264, August 2005.

[22] Dave Katz and Dave Ward. Bidirectional Forwarding Detection. RFC 5880, June 2010.

[23] Praveen Kumar, Yang Yuan, Chris Yu, Nate Foster, Robert Kleinberg, Petr Lapukhov, Chiun Lin Lim, and Robert Soulé. Semi-oblivious traffic engineering: The road not taken. In *Proceedings of USENIX NSDI*, pages 157–170, April 2018.

[24] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[25] Hongqiang Harry Liu, Srikanth Kandula, Ratul Mahajan, Ming Zhang, and David Gelernter. Traffic engineering with forward fault correction. In *Proceedings of ACM SIGCOMM*, pages 527–538, August 2014.

[26] Peter Phaal and Marc Levine. sFlow version 5, July 2004.

[27] Yakov Rekhter, Tony Li, and Susan Hares. A Border Gateway Protocol 4 (BGP-4). RFC 4271, January 2006.

[28] Eric C. Rosen, Arun Viswanathan, and Ross Callon. Multiprotocol label switching architecture. RFC 3031, January 2001.

[29] Rachee Singh, Sharad Agarwal, Matt Calder, and Paramvir Bahl. Cost-effective cloud edge traffic engineering with Cascara. In *Proceedings of USENIX NSDI*, pages 201–216, April 2021.

[30] Rachee Singh, Nikolaj Bjørner, Sharon Shoham, Yawei Yin, John Arnold, and Jamie Gaudette. Cost-effective capacity provisioning in wide area networks with Shoofly. In *Proceedings of ACM SIGCOMM*, pages 534–546, August 2021.

Detecting Ephemeral Optical Events with OpTel

Congcong Miao[1], Minggang Chen[1], Arpit Gupta[2], Zili Meng[3], Lianjin Ye[3], Jingyu Xiao[3],
Jie Chen[1], Zekun He[1], Xulong Luo[1], Jilong Wang [3,4,5], Heng Yu[3]

[1]*Tencent,* [2]*UC Santa Barbara,* [3]*Tsinghua University,* [4] *BNRist,* [5] *Peng Cheng Laboratory*

Abstract

Degradation or failure events in optical backbone networks affect the service level agreements for cloud services. It is critical to detect and troubleshoot these events promptly to minimize their impact. Existing telemetry systems rely on arcane tools (e.g., SNMP) and vendor-specific controllers to collect optical data, which affects both the flexibility and scale of these systems. As a result, they fail to collect the required data on time to detect and troubleshoot degradation or failure events in a timely fashion. This paper presents the design and implementation of OpTel, an optical telemetry system that uses a centralized vendor-agnostic controller to collect optical data in a streaming fashion. More specifically, it offers flexible vendor-agnostic interfaces between the optical devices and the controller and offloads data-management tasks (e.g., creating a queryable database) from the devices to the controller. As a result, OpTel enables the collection of fine-grained optical telemetry data at the one-second granularity. It has been running in Tencent's optical backbone network for the past six months. The fine-grained data collection enables the detection of short-lived events (i.e., ephemeral events). Compared to existing telemetry systems, OpTel accurately detects $2\times$ more optical events. It also enables troubleshooting of these optical events in a few seconds, which is orders of magnitude faster than the state-of-the-art.

1 Introduction

Cloud service providers, such as Google, Microsoft, and Tencent, have embraced the approach of setting up as many data centers as possible across metro areas [6, 20, 21, 23, 26, 38]. Such an approach enables cloud providers to physically get closer to the end-users, which in turn enables a wide range of applications with diverse bandwidth and latency requirements [29, 45]. The optical backbone network that interconnects these geographically distributed data centers is critical for ensuring reliable exchange of terabits of data every day [2, 3, 24, 25, 27]. Under the hood, the optical back-bone network is composed of optical hardware (e.g., optical transponders, amplifiers, wavelength (de-)multiplexers), and fiber cables. Degradation or failure of any of these components (i.e., optical events) would degrade the inter-DC connectivity, which in turn affects the service level agreements (SLAs) for cloud services [5, 18, 20, 49]. Therefore, to improve the reliability and availability of the optical backbone network, it is critical to promptly detect and troubleshoot optical events.

Unfortunately, existing telemetry systems are not designed for such fast-paced detection and troubleshooting of optical events. More concretely, they collect sampled or aggregated data from optical devices. Such coarse-grained data is not suited for either detecting short-lived optical events or troubleshooting related optical events to various stakeholders (i.e., application developers, data center tenants, etc.). Figure 1(a) illustrates the limitations of existing telemetry systems. Here, when a customer reports degradation in the quality of experience for video streaming service (e.g., rebuffering), attributable to a short-lived optical event lasting few tens of seconds. The network operator that looks at the telemetry data collected by the existing telemetry systems at the 15-minute granularity cannot detect or troubleshoot such a short-lived optical event. The current telemetry systems are slow in detecting and troubleshooting the more disruptive persistent events as well. Network operators need to query data from multiple vendor-specific controllers to stitch a holistic view of the underlying network, which is tedious and prone to errors. Our analysis of the trouble tickets dataset shows that it takes hours to days to troubleshoot the optical hardware failures. Though we witnessed the development of various network telemetry systems, such as Sonata [19], Marple [33], PathDump [42], OmniMon [22], etc., that offer packet-level network streaming analytics at scale, they are not suited for diagnosing degradation or failure events in optical networks.

The limitations of the existing telemetry systems are attributable to three key factors. First, the optical backbone network uses devices from multiple vendors (i.e., vendor-free optical systems in § 2.1), and the current telemetry systems develop interfaces for vendor-specific controllers to ac-
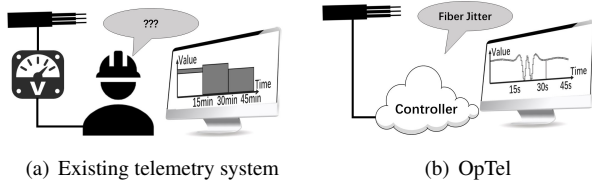
(a) Existing telemetry system   (b) OpTel

Figure 1: **(a) Existing telemetry systems fail to detect ephemeral events and are slow in detecting and troubleshooting persistent events. (b) OpTel detects and troubleshoots both ephemeral and persistent events promptly with the one-second granularity data.**

cess the optical data. Though vendor diversity is critical for cloud providers to deter vendor monopolies and avoid concurrent failures, *fragmented design* of existing telemetry systems is undesirable. It inhibits accessing optical data directly or extracting a consistent network view. Second, the existing telemetry systems rely on *arcane tool*, i.e., SNMP, to collect data from different devices. SNMP performs various data-management tasks, such as creating a local MIB database [35], supporting read and write operations to this database, etc., locally on the optical devices. Both faster reads (queries) and writes to this database will cause higher CPU usage. Given the limited resources at the device, it is not possible to query this data at higher frequencies with the SNMP protocol. Third, the vendor-specific controllers run on physical servers with fixed compute and memory resources. Such *inelastic resource allocation* for the existing telemetry pipelines creates multiple bottlenecks with the increasing number of optical devices or data-collection frequencies.

In this paper, we present the design and implementation of OpTel (Figure 1(b)), an optical telemetry system for optical networks. The proposed system offers direct access to optical data in a vendor-agnostic manner and offloads data-management tasks from the optical devices to cloud-based controllers that can easily scale with network size and collection frequency. We highlight the salient feature of the proposed system below.

Vendor-agnostic centralized control. OpTel shunts away vendor-specific controllers and replaces them with a single centralized controller that directly interfaces with optical devices in a vendor-agnostic manner. To enable such a vendor-agnostic design, we develop a standardized model for optical devices. This device-level model consists of two essential parts: logic and data model. Here, the logic model identifies key components common across devices from different vendors and standardizes their workflow. The data model specifies the configurable parameters for each component.

Streamline telemetry pipeline at optical devices. OpTel replaces SNMP (pull-based) protocol with a "push-based" telemetry pipeline. More concretely, it offloads the compute-intense data-management tasks from the optical devices to cloud-based controllers, with access to an elastic pool of re-

sources. Such streamlining of the telemetry pipeline offloads resource-intense operations to the cloud, enabling OpTel to collect fine-grained optical data at higher frequencies from resource-constrained optical devices. The telemetry pipeline at optical devices consists of the following key parts: telemetry manager, telemetry agent, cache, and aggregator. Here, the telemetry manager interfaces with the centralized controller and is responsible for receiving configurations from the controller and configuring other parts. The telemetry agent reads data from different modules and stores them into the local cache. The aggregator is responsible for pushing the data in the cache to the centralized controller.

The rest of the paper presents the background and motivation in Section 2, details the design and implementation in Section 3. We demonstrate how OpTel enables collecting fine-grained telemetry data at the one-second granularity and how such a dataset empowers network operators to promptly detect and troubleshoot optical events, both persistent and ephemeral, in Section 4. We have been running OpTel in Tencent's optical backbone network for the past six months. We report our experience of collecting and analyzing the telemetry data at scale. Notably, we demonstrate that access to such fine-grained data enables us to establish temporal relationships between different optical events.

2 Background and Motivation

We first provide an overview of the optical backbone network (§ 2.1). We then discuss why existing telemetry systems fail to promptly detect and troubleshoot optical events (§ 2.2).

2.1 Optical Backbone Network

The optical backbone network interconnects different data center sites, carrying terabytes of traffic each day. Figure 2 zooms-in into a specific link (i.e., an optical transport system) interconnecting two data center sites. Each link consists of an optical line system (OLS) and multiple optical transponder units (OTUs). Each OTU receives the electrical signal from the data center router (DR), and converts it into a specific wavelength, called an optical *channel*, and vice versa. When router ports have a lower capacity than the optical channel, the OTU encapsulates and multiplexes multiple router ports onto the channel.

The OLS contains two optical *segments*, one for each direction of network traffic. Each segment carries multiple optical channels, with wavelength division multiplexing/demultiplexing (MUX/DMUX) combining/splitting these channels and booster amplifier (BA) at the transmitting end and preamplifier (PA) at the receiving end. Segments also have in-line amplifiers (LAs) that amplify the signal in the optical domain to deal with long-haul transmission loss. Each part of the segment is called a *span*. As a special case, segment yields span if the OLS does not have LA. Optical Supervisory
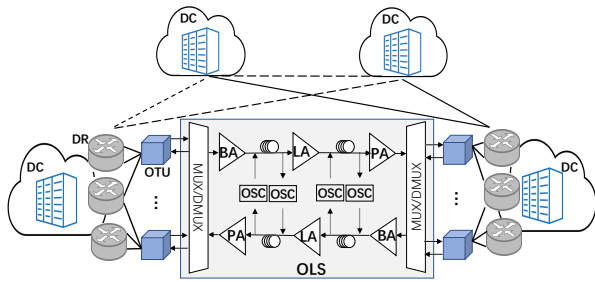
Figure 2: **An overview of the optical backbone network.**

Figure 3: **SNMP's data collection workflow**

(a) Performance of devices (b) Performance of controllers

Figure 4: **Performance of optical devices and vendor-specific controllers from two vendors.**

Channel (OSC) is an additional channel that does not carry any payload traffic and monitors each span.

Most cloud providers use optical devices from multiple vendors. Vendor diversity is intentional to deter vendor monopolies and avoid concurrent failures. Typically, cloud providers, including Tencent, embrace a *vendor-free design* of the optical transport system (i.e., vendor-free optical system), where they purchase optical line systems and optical transponder units from different vendors [11].

2.2 Monitoring Optical Backbone Network

Any degradation or failure events in the optical backbone network can affect the SLAs for various cloud services. Thus, it is critical for network operators to promptly detect and diagnose such optical events, which in turn requires collecting fine-grained optical data from the underlying optical devices at high frequency. The existing telemetry systems are not designed to support such intense data-collection requirements. We identify three key factors that inhibit existing telemetry systems to scale flexible data collection.

Highly fragmented design. The control plane for most optical backbone networks is highly fragmented as it relies on vendor-specific controllers to manage individual devices. The existing telemetry systems inherited this fragmented design, where a centralized controller interfaces with vendor-specific controllers to collect the required telemetry data. Such a fragmented design inhibits flexible and direct access to fine-grained optical data at scale. Each vendor-specific subsystem implements its workflow to collect the data from individual devices, affecting how frequently each subsystem reports the telemetry data. Additionally, the data schemas across vendors are different, which further inhibits supporting a consistent representation of the collected data.

To illustrate the impact of each of these factors, we use the metric, *polling delay*, which measures the difference in time when the centralized controller sends the poll request to vendor-specific controllers and when it receives the requested data. We have performed the measurement studies of two subsystems provided by vendor 1 and vendor 2. For confidentiality, we omit the vendor name. We observe it takes about 3 minutes and 7 minutes to complete the collection of 5 indicators from vendor 1 and vendor 2 respectively. Here, each indicator represents the type of data, such as `SNR`, `Q-factor`,
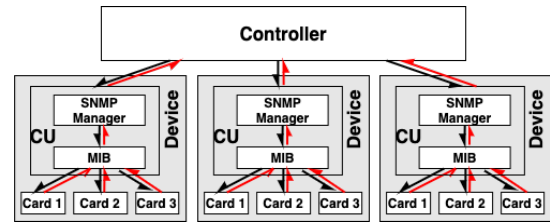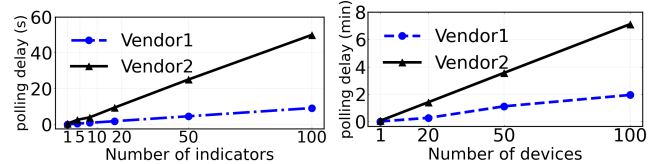
etc., collected from the devices. The difference in polling delay across two vendors is attributable to an artifact of different data-collection workflow each applied within its subsystem. Such high variance in polling delays across different vendors makes it hard for network operators to extract a consistent (synchronized) view of the network, affecting their ability to troubleshoot various optical events.

Reliance on arcane data-collection tools. Most existing telemetry systems for optical backbone networks rely on SNMP [10], which is not suited for high-frequency data collection. SNMP performs various data-management tasks locally on the optical device. More concretely, it creates and updates a local queryable database (MIB) on the device, and handles controller's queries. Figure 3 shows SNMP's data collection workflow. Here, to simplify exposition, we divide the optical device into control and data plane. Here, the control plane consists of SNMP manager and MIBs and the data plane comprises of multiple line cards. The black and red arrows represent the control and data flows respectively. Once the SNMP manager receives an SNMP GET request from the controller, it traverses the table in MIB database [35] one by one to get the `function` to obtain the data from the line card and then reports the requested data. This process is slow and consumes a significant number of CPU cycles, making it difficult to scale data collection frequency with SNMP.

Figure 4(a) shows how polling delay changes as the number of indicators increases. We observe that the relationship between polling delay and the number of indicators is linear. Our interactions with vendors revealed that this linear relationship is attributable to their choice of serializing read request for multiple indicators, reading only one at a time. This design choice limits SNMP's CPU usage, which competes with device's data-plane operations. Such a long polling delay with SNMP inhibits existing telemetry systems to collect fine-grained optical data at higher frequencies.
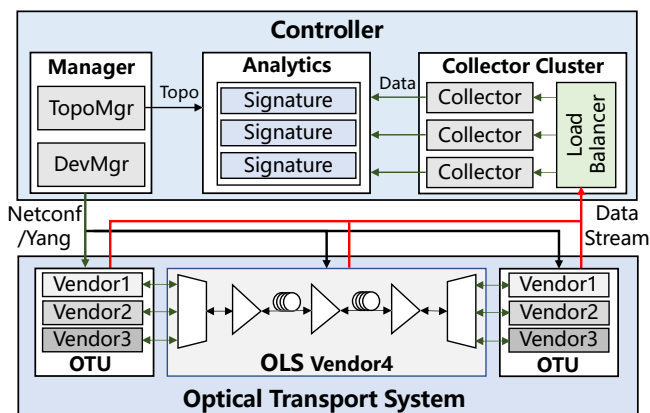
Figure 5: **Architecture of OpTel.**

Inelastic resource allocation for the telemetry pipeline.
Telemetry systems need to concurrently collect data from
all the underlying optical devices to ensure a consistent and
fine-grained view of the network. However, we observe that
the vendor-specific controllers run on physical servers with
fixed compute and memory resources. Such an inelastic de-
sign makes these controllers a bottleneck in existing telemetry
pipelines as the number of optical devices or the collection
frequency increases.

Unsurprisingly, we observe a linear relationship between
polling delay and the number of devices in Figure 4(b). This
behavior is also attributable to vendors' choice to serialize
requests at the controller. Such serialization ensures that the
controller can handle all the incoming requests with a fixed set
of resources at the cost of longer polling delays, which affects
the ability to construct a consistent view of the network at
fine time scales. More concretely, it is impossible to correlate
the optical data across two different optical devices on a short
time scale, affecting the troubleshooting capabilities of the
existing telemetry systems.

3 OpTel's Design and Implementation

We now describe how the proposed system, OpTel, addresses
the limitations of existing telemetry systems described above.
We first state its design goals in Section § 3.1, and then de-
scribe how it achieves these goals in Section § 3.2 and § 3.3.

3.1 Design Goals

OpTel's goal is to extract multiple indicators from all the
devices in the optical backbone at finer time granularities, i.e.,
order of seconds. Such a dataset is critical for timely detection
and diagnosis of various disruptive events in the backbone
network. OpTel addresses the limitations of existing telemetry
systems to achieve this goal. More concretely, to address the
fragmentation issue, it bypasses vendor-specific controllers to
collect the telemetry data directly from the optical devices in
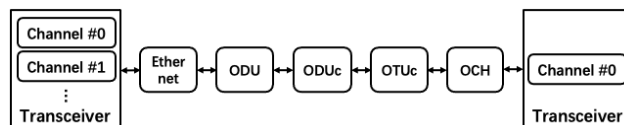a vendor-agnostic manner. To address the scalability issues, it

Figure 6: **The logic model of OTU.**

streamlines the telemetry pipeline such that it performs all the
complex data-management tasks to a centralized controller
running in the cloud. Such a design ensures that the data
collection pipeline is not bottlenecked by limited compute
resources at the individual devices. The centralized controller
has access to an elastic pool of resources in the cloud.

3.2 Vendor-agnostic Centralized Control

Figure 5 presents OpTel's architecture. Here, the centralized
controller directly interfaces with the optical devices in a
vendor-agnostic manner. We developed a standardized model
that abstracts away the vendor-specific details for the con-
troller. We now describe how we develop the vendor-agnostic
device model and how it enables collecting data directly from
the optical devices.

3.2.1 Standardized Model for Optical Devices

In vendor-free optical systems, the operation performed by
different optical devices is similar at a high level, but the
specific logic and workflow vary across vendors. Such het-
erogeneity across devices from different vendors complicates
the design of vendor-agnostic interfaces. We develop a stan-
dardized model for optical devices that abstracts away the
vendor-specific details to address this challenge. It consists of
two parts: *logic model* and *data model*. Here, the logic model
identifies key components common across devices from differ-
ent vendors and standardizes their workflow. The data model
specifies the configurable parameters for each component.
Logic model. The first challenge in developing vendor-
agnostic interfaces is that the physical components and their
workflow are proprietary to each vendor. To address this chal-
lenge, the logic model first identifies a group of logical compo-
nents that are common across devices from different vendors.
It then standardizes the workflow between these components.
To illustrate, consider the case of optical transponder units,
i.e., OTUs. Figure 6 shows OTU's logic model. Here, the
logic model first identifies four logical components across all
vendors: Ethernet, optical data unit (ODU, ODUc), optical
transport unit (OTUc), and optical channel (OCH). Recall that
an OTU encapsulates and multiplexes multiple router ports
onto an optical channel. The ODUc is a high-order data unit
after combining the payload data from multiple router ports.
The logic model then specifies the workflow between these
components. For example, the mapping between Ethernet and
ODU represents an encapsulation of an Ethernet frame into
an ODU frame. Such an abstraction enables the standardized
representation of different optical devices.

Data model. The second challenge in enabling vendor-agnostic interfaces is that the capability of physical components inside the device is different across vendors, although their functions are the same. For example, the range of gain of an optical amplifier provided by vendor 1 is 15-25 dB, while it might be 20-30 dB from vendor 2. This heterogeneity complicates managing these devices in a vendor-agnostic manner. We design a component data model with specific descriptions of configurable parameters of each component. When each device connects to the controller, the controller obtains the specification datasheets from the device and initializes the corresponding value of configurable parameters. Such an approach simplifies the management complexity of heterogeneous devices, regardless of the capability of physical components inside the device.

We have developed a model for each device type for our optical backbone network. Our experience using these models in production settings was smooth, demonstrating their generalizability.

3.2.2 Centralized Data Collection

The standardized model allows the centralized controller to access the telemetry data directly, enabling OpTel to shunt away vendor-specific controllers. The centralized controller consists of three key modules: *global manager*, *scalable collector* and *real-time analytics*, to perform detecting and troubleshooting optical events at scale in a timely manner.

Global manager. It consists of two parts: device manager (DevMgr) and topology manager (TopoMgr). The DevMgr is responsible for configuring the underlying optical devices. For each device, it leverages the relevant standardized model to configure devices in a vendor-agnostic manner. It completes this process by issuing a Yang file [7] to the device through the vendor-neutral Netconf protocol [13]. The TopoMgr maintains a physical topology of optical devices to provide a network-wide view of the optical networks and thus helps the real-time analytics to troubleshoot the optical events at scale. To illustrate how TopoMgr aids troubleshooting, consider the case of degradation in a fiber cable. Here, as it is not possible to directly collect the data from the cable, the analytics can instead use the TopoMgr to identify the two terminal devices at each end of the cable. It can then query the transmit (Tx) and receive (Rx) power data from these devices for troubleshooting.

Scalable collector. This module is a cluster of multiple collector nodes designed to handle changes in the number of indicators, collection frequency, or the number of optical devices over time. With the aid of the cloud's elastic pool of resources, it can scale horizontally by adding (or removing) collector nodes over time. It relies on a load balancer to distribute the load among individual collectors within the cluster. It is robust against the failure of a particular collector node.

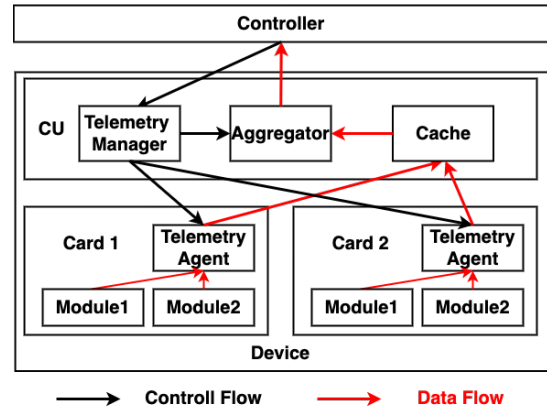**Real-time analytics**. It performs the task of promptly de-

Figure 7: **Push-based optical telemetry**

tecting and troubleshooting optical events by combining the optical data from the collector and the topology information from the global manager. The workflow of real-time analytics consists of two parts: detection and troubleshooting. To detect degradation or failure events, it monitors the values of optical data in real-time and raises the alarm if the value exceeds a pre-specified threshold. In parallel, it starts the troubleshooting process. Rather than manually troubleshooting the optical event, it leverages the signatures of previous optical events for diagnosis. To illustrate, consider the case when the received optical power becomes zero for a device. The analytics module first raises the alarm with a message, *the receiver can not receive the light* and then begins troubleshooting. It matches the collected data with previous signatures. If it finds the match, it simply sends troubleshooting report to the operator. If the collected data does not match a pre-existing signature, it lets operators manually express their queries for troubleshooting. It automatically updates the relevant signatures for future events. Our deployment experience shows that is possible to troubleshoot most of the optical events using existing signatures.

3.3 Streamlined Telemetry Pipeline

Promptly detecting and troubleshooting optical events requires collecting fine-grained data from the underlying optical devices. The widely used SNMP is flawed in performance because it performs various data-management tasks locally on the resource-constrained optical devices (Figure 3). In contrast, OpTel offloads compute-intense operations from the optical devices to the centralized controllers by push-based telemetry pipeline, enabling to collect the fine-grained optical data at higher frequencies. Figure 7 depicts the architecture of push-based optical telemetry. The telemetry pipeline at optical devices consists of the following key parts: *telemetry manager*, *cache* and *aggregator* in the control unit (CU), and *telemetry agents* in the line cards. The telemetry manager is responsible for the configurations of other parts, i.e., telemetry agent and aggregator. The telemetry agent reads data from different modules and stores them into the local cache. The

aggregator is responsible for pushing the data in the cache to the centralized controller. In the following, we will describe them in detail.

Telemetry manager. The offloading of compute-intense data-management tasks from the optical devices to the controller requires preliminary configurations at the device. The telemetry manager firstly interfaces with the centralized controller to obtain the YANG file [7] and then parses the YANG file to configure the telemetry agent and aggregator. The aggregator is configured to periodically initiate a connection to push the optical data from the local cache to the controller. As for the telemetry agent in the line card, it is configured in three parts: the destination of data (i.e., cache), the source of data (i.e., modules in the line card), and the periodicity that the telemetry agent should push the data.

However, based on the real-world deployment experiences, we observed that configuring the same periodicity for pushing data at the telemetry agent and aggregator may result in the frequent data loss in the controller. This phenomenon is attributable to the different timing mechanisms. Generally, the CU always runs a Linux operating system and enables network time protocol (NTP) [30] to keep timing. However, some line cards are the embedded equipment without running a Linux operating system, resulting in it being unable to keep timing through NTP. Thus, these line cards keep timing through the crystal oscillator. The frequency deviation inside the crystal oscillator will lead to the timing inaccuracies [44]. Therefore, the performance data pushed by the telemetry agent is not strictly periodic. Slower timing will result in the data not being stored in the local cache, which in turn causes data loss in the controller. For example, assume that the controller needs to collect the data from the device at the one-second granularity. The telemetry manager configures the telemetry agent and aggregator to push the data every one second. However, the frequency deviation inside the crystal oscillator may result in the timing in the line card slower than that in the CU. It will take more than one second for the telemetry agent to push the data to the local cache. Therefore, the aggregator will push the empty data to the controller. Motivated by this, we always configure the data pushed in the telemetry agent at a higher frequency.

Telemetry agent. Once configured, the telemetry agent periodically performs the card-level data collection through the vendor-specific protocols and pushes the data to the local cache. Specifically, the values of data are generated in two ways: instant value and accumulated value.

Instant Value. It is a sampled data in a given time interval. The receiver captures the physical analog signal and then translates it into the digital value, which is further stored in the RAM. Figure 8(a) describes the process of generating instant value of the received signal in the physical layer. The PIN photodiode firstly captures the light signal and transforms it into the analog current. An analog-to-digital converter (ADC) is applied to convert the analog current into a digital value

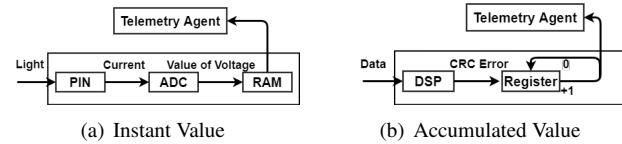(a) Instant Value (b) Accumulated Value

Figure 8: **The process of generating specific values. (a) Instant value records the performance in the physical layer; (b) Accumulated value records the performance in the data link and network layer.**

of voltage which is further stored in the RAM. The telemetry agent periodically reads RAM to collect the data through the vendor-specific protocol. Note that the value in RAM will be replaced frequently, thus enables the data to be collected at higher frequencies. In our work, the instant value records the performance in the physical layer. We use transmit/receive (Tx/Rx) power to detect optical events, and signal-to-noise ratio (SNR) and quality factor (Q-factor) to check the ability of the optical system to transmit data (Figure 9(a) and 9(b)).

Accumulated Value. It is a counting value accumulated across the whole timeline. The digital signal processor (DSP) processes the received digital signal and counts in a certain way. Figure 8(b) describes the process of generating the accumulated value of the CRC error. The register counts the volume of CRC errors in the whole time interval. The telemetry agent periodically reads the register to collect the value. After that, the register will be reset to its initial value. The accumulated value records the performance in the data link and network layer, such as CRC error and post forward error correction (FEC). We use them to differentiate optical events according to the influence of optical events in the data link and network layer (Figure 9(c),9(d)).

Cache. The local cache serves as data storage that stores the performance data received from the telemetry agent and then bundles data at the device level. It is compatible with the performance data pushed by the different agents at different frequencies. Generally, the data for a single indicator stored in the telemetry cache is more fine-grained since the frequency of the telemetry agent pushing the data is higher than that of the aggregator reading the data. The data in the local cache will be cleaned after being read.

Aggregator. The aggregator periodically initiates a connection to get the bulked data from the local cache. Since the data provided by the cache is more fine-grained, the aggregator should merge the data to get representative statistics and push them to the controller through the gRPC protocol [1].

4 Evaluation

OpTel has been running in Tencent's backbone network for the past six months, demonstrating its deployability in production settings. In this section, we show how the proposed streamlined telemetry pipeline enables collecting all possible indicators from all the optical devices in the network at the

one-second frequency (Section 4.2). We then show how such fine-grained data enables the detection of ephemeral optical events (Section 4.3). We investigate how ephemeral events help predict more disruptive future events, illustrating the utility of such a fine-grained telemetry system (Section 4.4). We also demonstrate how such fine-grained data enables troubleshooting optical events in the order of few seconds, which is orders of magnitude faster than possible with the existing telemetry systems (Section 4.5).

4.1 Setup

4.1.1 Dataset

We use OpTel to curate three datasets. Here, we collect the data for six months (July-December, 2020) from Tencent's optical backbone network. This backbone has O(50) links, O(100) spans, O(100) segments, O(1000) optical channels, and O(1000) optical devices from O(10) vendors. For confidentiality reasons, we do not report the exact numbers.

Optical telemetry dataset. We curate this dataset by collecting all indicators from all the optical devices at one-second granularity using OpTel. We collect the Tx/Rx power levels, SNR, and Q-factor from the physical layer. From the data link layer, we collect the Post Forward Error Correction Bit Error Rate (FEC BER) [31], loss and error frame rate (i.e., the ratio of the number of Rx vs. Tx frames and Error vs. Rx frames). We also collect cyclic redundancy check (CRC) error rate [40] from the network layer. Here, the physical layer indicators are "instant" values, and the rest are the "accumulated" values (§ 3.3). Also, note that since the OTU encapsulates and multiplexes payload from router ports, we collect the data link and network layer indicators directly from the OTU (§ 3.2.1). However, it is not efficient to only focus on the Tx/Rx power in OTU or BA/PA to troubleshoot optical events. For example, if there are several spans and LAs in a segment, and the Rx power of PA becomes 0 while the Tx power of BA does not change, we can not distinguish which span is responsible for the event. Thus, we combine the Tx/Rx power of OSC for span-level monitoring. The detailed origins of telemetry data are shown in Figure 17 in appendix A.

Location dataset. We use OpTel's TopoMgr to curate this dataset. It maintains a topology of the devices to provide a network-wide view to establish a relationship between different devices. Such relationships are critical for troubleshooting as indicators from a single device are often not enough to diagnose various optical events. For example, diagnosing degradation events in fiber cables requires data from both ends of the fiber cable.

Trouble tickets dataset. We collect this data from the network management platform at Tencent. We first filter out the events related to the optical networks (see appendix B for details) and then categorize these optical events into a small number of classes, i.e., fiber cable, hardware, and power
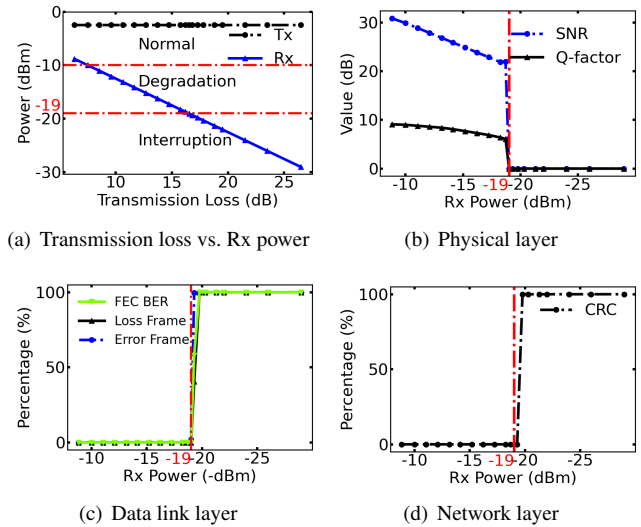

(a) Transmission loss vs. Rx power   (b) Physical layer

(c) Data link layer   (d) Network layer

Figure 9: **An example of the physical/data link/network layer behaviors with the increase of transmission loss.**

events. Each ticket contains a timestamp recording the event's start time with detailed messages and a corresponding timestamp recording the localization of the event, i.e., event name (e.g., optical fiber jitter, amplifier instability, etc.). Note, troubleshooting optical events requires much manual effort in existing telemetry systems. We use this data to learn signatures of different optical events and show the time efficiency of OpTel on troubleshooting optical events by comparing it with the existing telemetry system.

4.1.2 Optical Events

We now present how we categorize optical events on the basis of their impact (degradation vs. interruption events) and duration (ephemeral vs. persistent events).

Interruption vs. Degradation events. To categorize optical events on the basis of their impact, we investigate the relationship between indicators at the physical, data link, and network layer (see Figure 9). Specifically, we take an optical transport system as an example, and fix the Tx power and iteratively adjust the transmission loss of optical fiber to simulate the degradation/failure event. Figure 9(a) shows a linear relationship between the transmission loss and values of Rx power in OTU based on the formula *Rx power (dBm) = Tx power (dBm) - transmission loss (dB)*. We observe similar trends in PA (not shown for brevity). After establishing the relationship between transmission loss and Rx power, we study how degradation in the power level at the receiver affects SNR and Q-factor at the physical layer (Figure 9(b)); FEC BER, loss frame and error frame rate at the data link layer (Figure 9(c)); and the CRC at the network layer (Figure 9(d)).

For higher Rx power levels (i.e., around -9 dBm), the values of SNR and Q-factor are high with SNR=31 dB and Q-factor=9 dB. The SNR and Q-factor indicate the ability of the
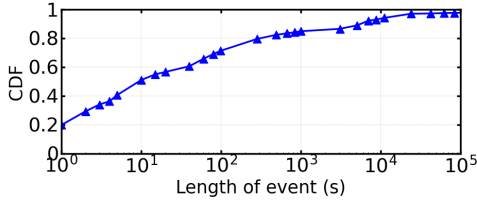
Figure 10: **The CDF of optical events' duration.**

Table 1: **The proportion of four types of optical events.**

Type	P-I	P-D	E-I	E-D	Total
Percentage	44.63%	4.28%	16.85%	34.24%	100%

(a) Collection frequency

(b) Number of indicators
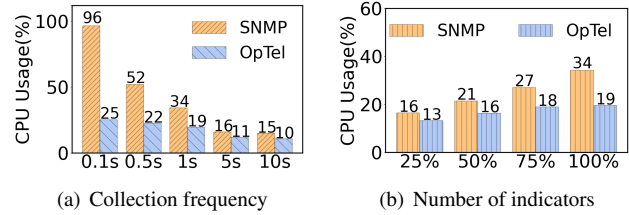
Figure 11: **CPU usage of the device with different collection frequencies and numbers of indicators (normalized).**

system to transmit data. Higher values for these physical layer indicators imply higher possibility of correctly decoding the transmitted '1's and '0's signal, and vice versa [18]. As Rx power decreases, SNR and Q-factor decrease linearly. But the values of data link and network layer indicators do not change as Rx power level above a specific threshold guarantees correct decoding of the transmitted signals. When the Rx power is below the threshold (i.e., -19 dBm), the Q-factor and SNR are below the sensitivity of transceiver and thus, it reports 0 to represent the abnormal state of optical system. For links with low SNR and Q-factor, the post-FEC BER increases because the number of error bits exceeds its error correction capability. Consequently, the receiver can not restore the transmitted data, resulting in nearly 100% of frame loss and error in the data link layer (Figure 9(c)) and packet loss due to CRC error in the network layer (Figure 9(d)).

Given these observations, we conclude that the Tx/Rx power level is the key indicator of optical-layer performance. This reinforces the prior study [53]. Thus, We use the changes of Tx/Rx power level to detect optical events and divide optical events into two broad categories on the basis of their impact: *degradation* and *interruption*. Generally, there is a conservative deployment of the optical transport system, with redundancy baked in at the Rx power. The degradation event occurs when the optical system transitions to an abnormal state, evident from smaller values for physical layer indicators Tx/Rx power level, Q-factor, SNR, etc. However, here such anomalies do not affect the data transmission at the data link or network layer. In contrast, the interruption events are where further degradation in the physical layer starts affecting data transmission. Note that fluctuations in Rx/Tx power levels are common in production networks. Based on the network operator's experiences, we treat any fluctuation within 1 dB range as normal.

Ephemeral vs. Persistent events. Optical events not only vary in terms of impact but also in duration. Figure 10 shows the duration of optical events (both interruption and degradation). We observe that the event duration exhibits long-tail behavior. Interestingly, we observe that 20% of events only last for one second and more than 50% of them last for less than ten seconds, indicating the prevalence of such transient optical events in the optical backbone. These observations demonstrate the utility of OpTel's ability to detect such short-lived events that go unnoticed with the existing telemetry systems. Given these observations, we divide optical events into two categories based on their duration. We call all the

optical events that last less than ten seconds as *ephemeral events* and the rest as *persistent events*.

Overall, we consider four different types of optical events based on the combination of their impact and duration: *ephemeral degradation (E-D)*, *persistent degradation (P-D)*, *ephemeral interruption (E-I)* and *persistent interruption (P-I)*. Table 1 shows the prevalence of each of these event types in our dataset. For confidentiality, we do not report the exact numbers. We observe that optical events of the type P-I contribute 44.63% to the total, followed by the E-D events, which contribute about one-third to the total. The P-D events are the least prevalent, only contributing 4.28% to the total events. Note that more than 50% of optical events are ephemeral.

4.2 Data Collection Overheads

Intuitively, we expect collecting optical data at higher frequencies (i.e., order of seconds) to be prohibitively expensive. We now demonstrate how OpTel's streamlined telemetry pipeline makes such high-frequency data collection feasible. We compare OpTel's overhead, quantified in terms of CPU usage at the optical devices, with existing SNMP-based telemetry systems for different collection frequencies (i.e., 0.1s, 0.5s, 1s, 5s, and 10s) and the number of indicators (i.e., 25%, 50%, 75%, and 100% of the total). We can configure the same device to either use OpTel's or conventional SNMP-based telemetry pipelines in our current deployment. Such flexibility enables us to report the CPU usage for these two different pipelines for the same set of optical devices.

Figure 11(a) shows that CPU usage increases with collection frequency for both pipeline, but the rate of change for OpTel is marginal. More specifically, the increase in collection frequency from 1 second to 0.1 seconds raises SNMP-based pipeline's CPU usage from 34% to 96%. Such high CPU usage highlights SNMP's struggles to handle polling requests at such high frequencies. In contrast, OpTel's CPU usage only increases by 6% from 19% to 25%, demonstrating its efficacy. As shown in Figure 3, the polling-based SNMP consumes a significant number of CPU cycles to collect data, including receiving the request from the controller, traversing the MIB
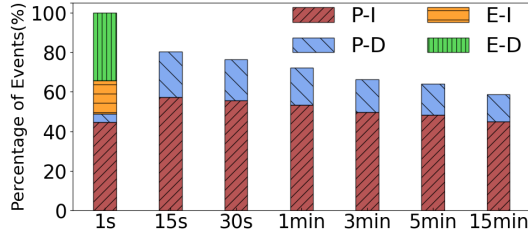
Figure 12: **The percentage of events detected with the decrease of collection frequencies (y axes is normalized by total events detected with one-second granularity data).**

Table 2: **The comparison of detected optical events with OpTel and the existing telemetry system.** *UND* **means undetected optical events.**

		Existing system (15 minutes)			
		P-I	P-D	*UND*	Total
OpTel (1 second)	P-I	**33.80%**	0	10.83%	44.63%
	P-D	0	**1.92%**	2.36%	4.28%
	E-I	11.00%	0	5.85%	16.85%
	E-D	0	11.88%	22.36%	34.24%
	Total	44.80%	13.80%	41.40%	100%

database [35], and then requesting data from the line cards. The reduction in the CPU usage of OpTel is attributable to the offloading of compute-intense data-management tasks from the optical devices to the controller. As shown in Figure 7, once configured, the device only needs to periodically initiate a connection to push the data to the controller. This process does not introduce much CPU overhead on the device.

Figure 11(b) shows that CPU usage increases with the number of indicators, but the rate of change for SNMP-based pipeline is greater than OpTel's. Specifically, the CPU usage is 16% if we collected 25% of total indicators with an SNMP-based pipeline. However, the CPU usage increases to 34% if all indicators are collected. In contrast, the CPU usage is only 19% for OpTel. Recall that vendors limit SNMP's CPU usage at the cost of longer polling delays. Figure 4(a) depicts that it takes tens of seconds to complete a polling period. The high polling frequency results in a new polling request starting before the previous polling request has ended. There will be a lot of concurrent polling requests, resulting in high CPU usage. The current design choice of SNMP is not scalable to collect a large number of indicators at higher frequencies. In contrast, OpTel streamlines the telemetry pipeline by offloading resource-intense operations to the cloud. Therefore, OpTel maintains low CPU usage at the device with the increasing collection frequencies and indicators.

4.3 Detecting Optical Events with OpTel

Detecting optical events is essential for troubleshooting the related network disruptions to various stakeholders. We evaluate the efficiency and accuracy of OpTel on detecting optical events by comparing with existing telemetry systems.

High Efficiency. We firstly study the efficiency of OpTel on detecting optical events by comparing with different collection frequencies, i.e., time intervals varying orders of minutes (1min, 3min, 5min, and 15min) and seconds (1s, 15s, and 30s). Previous works only took advantage of minute-level data to study operational optical networks [8, 18, 39, 52]. To the best of our knowledge, no prior work uses second-level data to detect optical events in operational optical networks. We introduce them in experiments to further demonstrate the relationship between the number of detected events and collection frequency. Specifically, we take the data collected at

the one-second granularity as the ground truth and simulate the detection of events with different collection frequencies.

Figure 12 demonstrates that OpTel achieves high efficiency on detecting optical events. For confidentiality, we do not report the detailed number of events. As the figure shows, the total number of detected events decreases when the collection frequency decreases. Specifically, OpTel outperforms the collection frequencies with 15 seconds, 1 minute, and 15 minutes by 25%, 39%, and 71%, respectively. This phenomenon proves the efficiency of OpTel to detect them. Another observation is that the collection frequencies lower than 15 seconds can not detect ephemeral optical events, and the number of persistent events (i.e., P-I and P-D) decreases when the collection frequency decreases. OpTel takes advantage of the one-second granularity data to exactly detect these ephemeral events. Surprisingly, we observe that the number of persistent events detected with the 15-second granularity data is more than that with the 1-second granularity data. This phenomenon implies that the majority of ephemeral events are wrongly identified as persistent events, i.e., E-I and P-I are wrongly identified as E-D and P-D, respectively. In other words, a portion of persistent events detected with the coarse-grained data are not actually persistent. This motivates us to learn the accuracy of detecting optical events by OpTel.

Full Accuracy. We then study the accuracy of OpTel on detecting optical events by comparing with existing telemetry systems. We take the existing system with 15-minute granularity data as an example since it is widely studied for optical layer in previous works [8,18,39,52]. Similarly, we regard the one-second granularity data as the ground truth and simulate the detection of optical events. The results are shown in Table 2. For confidentiality, the number of events is normalized by the total number of optical events detected with the 1-second granularity data. Each row represents the events detected by OpTel, while each column represents the events detected by the existing telemetry system with 15-minute granularity data. We observe that the existing system can only correctly detect 35.72% of optical events (shown in **Bold**), while 41.40% of optical events are not detected (*UND* column) and 22.88% of optical events are wrongly detected (shown in underlined). Specifically, for P-D events, only less than 50% of P-D events can be accurately identified by the existing telemetry system while the rest can not be detected. As for ephemeral events, they are either identified as persistent events or not detected
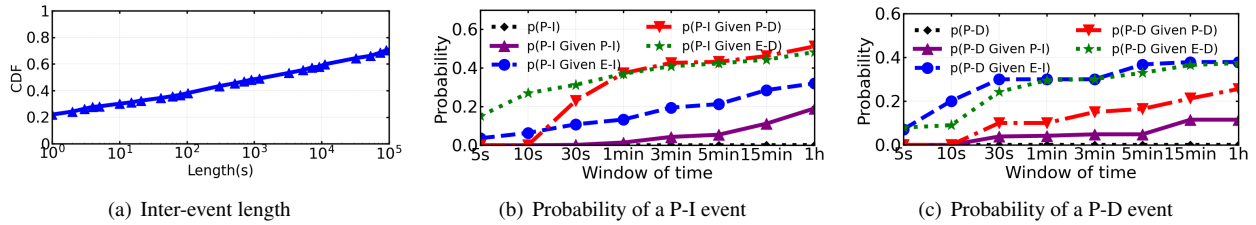
Figure 13: **(a) The CDF of length of inter-events (Log-scale x-axes); (b) Probability of a P-I event in a given time window after different types of events; (c) Probability of a P-D event in a given time window after different types of events.**

by the existing telemetry system. As for E-D, 22.36% of E-D events can not be detected, occupying about two thirds of total E-D events. It can be caused by several reasons. For example, if there are several E-D events in one 15-minute time interval, only one E-D event will be identified as a P-D event, and the rest can not be identified. In general, our OpTel with 1-second granularity data accurately detects all optical events, especially for ephemeral events.

4.4 Predicting Future Optical Events

We evaluate the possibility of predicting future events based on the current event within a short time by OpTel. Figure 13(a) depicts the CDF of length of inter-events. A surprising observation is that 20% of inter-event lengths are only one second. This phenomenon suggests that these events occur in bursts and demonstrates the utility of the optical telemetry system on collecting data at the one-second granularity. Another observation is that 50% of inter-event lengths are less than 1000 seconds, suggesting a high probability of an optical event within about 15 minutes after the current event.

We focus on predicting persistent events as they represent a more prolonged impairment or loss in network capacity and are more predictable. Taking the P-I event as an example, we first compute the probability of a persistent interruption event within a window of time and call it p(P-I). For $x \in$ {P-I, E-I, P-D, E-D}, p(P-I given x) indicates the probability of observing a P-I event given a prior x event within the same window. Fig 13(b) and 13(c) depict the average probabilities across all spans as a function of window size, from 5 seconds to 1 hour. In contrast to the previous work [17], our works focus on taking advantage of one-second granularity data and the ephemeral events to achieve the short-term predictions.

As expected, p(P-D) and p(P-I) increase as window size increases; the larger the window of time, the higher possibility of a persistent event occurs within that window. For a window of 1 hour, the probability of a persistent event occurrence is less than 1%. This suggests a low probability of having a persistent event in the 1-hour window. However, there is a significant jump in the probability if there has been another event in the past, e.g., E-D, E-I, and P-D. For example, for a window of 1 hour, the probability of persistent event occurrence increases to about 50% if there has been an E-D event within that window. Meanwhile, we observe that the events

have a strong relation in a short time window. For example, for a window of 5 seconds in Figure 13(b), the probability of P-I occurrence increases to about 20% if there has been an E-D event within that window, while for a window of 1 minute, the probability increases to 40%. This indicates that the E-D event is strongly related to the future P-I event. As for the prediction of P-D events in Figure 13(c), the probability of P-D occurrence increases to 30% if there has been an ephemeral event (i.e., E-D and E-I) within 30 seconds, and the possibility does not increase much with a larger window of time. This suggests that the P-D event always happens after the ephemeral event within 30 seconds. Another observation is that the past P-I event is less predictive of the future persistent events, indicating that the P-I events are memoryless.

OpTel demonstrates a high possibility for predicting future events at the second-level granularity. Thus, network operators could take the fine-grained IP layer network management. First, network operators should monitor the ephemeral and degradation events and raise alarms when they occur. Then, appropriate actions should be taken since the failure probability of IP layer link will increase. For example, they could improve traffic engineering so that important traffic should be dispatched away from the corresponding link.

4.5 Troubleshooting Events with OpTel

Characterizing failure signals. We demonstrate the effectiveness of OpTel on unveiling the signatures of optical events, and thus we could quickly troubleshoot the optical events based on the observed signatures. We use the Tx/Rx power as the primary method to unveil the signatures of optical events since it is the key indicator of optical-layer performance [53]. For some optical events such as fiber events, we learn the signatures based on the network operator's experience (Figure 14). However, for some optical events such as hardware failures, we should traverse the trouble tickets dataset to learn the signatures. To locate optical events accurately, we take advantage of the centralized controller to conduct inter-device analysis by combining the Tx/Rx power from three sources, i.e., OSC, OTU, and BA/PA. For confidentiality, we do not show all signatures of optical events and take an example for each category of the events.

(a) Fiber cable: optical fiber jitter before interruption (Figure 13(b)). We firstly unveil the most frequent optical
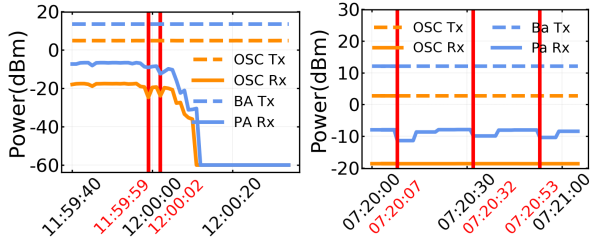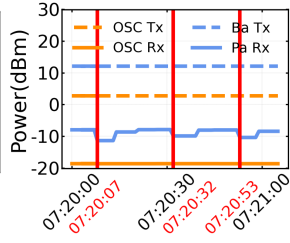
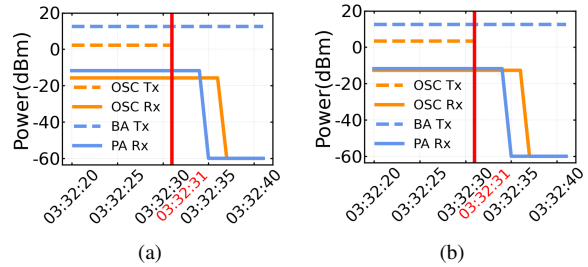Figure 14: **Fiber**   Figure 15: **Amplifier**

Figure 16: **The power outage at site of LA.**

events, i.e., fiber cable events. As shown in Figure 14, on the one hand, we observe that the Tx powers of BA and OSC remain unchanged during the timeline. However, the Rx powers at both PA and OSC is down to -60 dBm after 12:00:10 (The receiver records a predefined minimum value when the received power is below its sensitivity.). On the other hand, the timestamp of Rx power changes in OSC and PA is consistent with several ephemeral degradation events at 11:59:59 and 12:00:02 before the interruption. Thus, we can localize the optical event as an optical fiber event since the sources of the light work well and the probability of two receivers at PA and OSC having problems at the same time is relatively low.

(b) Hardware: amplifier instability. We then unveil one example of hardware failures, i.e., amplifier failure. Since the curve is quite similar, we only select a 1-minute time interval, as shown in Figure 15. We observe that the Rx power of PA changes periodically with about a 3 dB drop at 7:20:07, 7:20:32, and 7:20:53, while the indicators of the rest remain unchanged. The stable Tx/Rx values of OSC indicate a normal state of fiber cable, while the stable Tx values of BA indicate that BA works well. Thus, we can localize the optical event as the instability of PA, i.e., amplifier failure.

(c) Power: power outage at site of LA. We unveil one example of power events, i.e., the power outage at the site of the in line amplifier (LA). In a long-haul transmission system, there is a relay site containing LA that amplifies the signal to deal with long-haul transmission loss. Figure 18 in Appendix C depicts the detailed origins of Tx/Rx power in Figure 16. In Figure 16(a), We observe that the Tx power of OSC can not be collected after 03:32:31 while the Tx power of BA remains almost unchanged. Surprisingly, the Rx power values of both OSC and PA become -60 dBm after 03:32:32. Thus, we locate the power outage event at LA in the site, and the delay of Rx power is mainly due to energy storage of components in the device, such as capacitance and inductance [12]. The similar results in Figure 16(b) further prove this phenomenon. Thus, we localize the optical event as a power outage at site of LA.

These signatures of optical events present the necessity of indicators to be collected at the second-level granularity which can not be demonstrated in the existing telemetry system. OpTel unveils the signatures of optical events, which presents the superiority of optical telemetry to collect data at the one-second granularity and a centralized controller to conduct inter-device analysis in real time.

We finally evaluate the time efficiency on troubleshooting optical events by comparing OpTel with the existing telemetry system. For the existing system, it takes much manual effort to troubleshoot the optical events, and we calculate the total time of troubleshooting the optical events based on timestamps recorded in the trouble tickets dataset. As for OpTel, based on signatures learned before, OpTel conducts inter-device analysis in a centralized controller to detect and troubleshoot optical events in a timely manner. Table 3 presents the comparison of the total time of troubleshooting optical events between OpTel and the existing telemetry system across all event categories. We do not report events that have not happened in the studied dataset. There are several observations. Firstly, 89.7% of optical events (i.e., P-D, P-I, E-D, and E-I) are caused by fiber cable, including fiber cut, fiber jitter, and fiber bent/degradation. The existing telemetry system takes about 5min ~10min to troubleshoot a fiber cut event. However, it can not troubleshoot the optical events caused by fiber jitter or degradation. In contrast, OpTel only takes several seconds to troubleshoot all the events related to the fiber cable. Secondly, 7.8% of optical events are caused by hardware, and it takes quite a long time, i.e., hours ~days and much manual effort to troubleshoot them. Some hardware events, such as amplifier instability (Figure 15) can not be troubleshooted in the existing telemetry system. In contrast, OpTel only takes about 2s ~60s to troubleshoot all the events related to hardware, reducing the time by as much as two to four orders of magnitude. The total time of troubleshooting events by OpTel is related to the time length of the signature. For example, OpTel takes 2s to troubleshoot amplifier malfunction and 60s to troubleshoot amplifier instability since we need to take about 60s to get the value change patterns of Rx power in PA to troubleshoot the optical event (Figure 15). As for the power events, OpTel is efficient to troubleshoot these events within a minute. In summary, OpTel takes advantage of the one-second granularity data to learn the signatures of the optical events and thus troubleshoots optical events at scale in a timely manner.

5   Related Work

Network streaming telemetry. Previous works have extensively studied the design of network streaming telemetry systems. End-host-based network streaming telemetry sys-

Table 3: **The comparison of the total time of troubleshooting events between OpTel and the existing telemetry system.**

Event category	Percentage	Event type (Detect)	Event name (Troubleshoot)	Existing telemetry system	OpTel
Fiber cable	89.7%	PI	Fiber cut	5min~10min	10s
		EI / ED	Fiber jitter	UNK	3s
		PD	Fiber bent / degradation	UNK	10s
Hardware	7.8%	PI / ED	Amplifier malfunction / instability	hours~days / UNK	2s~60s
		PI / ED	OSC malfunction / instability	hours~days / UNK	2s~60s
		PI	OTU malfunction	hours ~days	2s~60s
Power	2.5%	PI	Power outage	hours	10s~30s
		PI	Power down	hours	10s~30s

tems [4, 16, 32, 41] performed flow-level tracking but had to deal with a limited view of the network. Switch-based network telemetry systems usually offered a coarse-grained view of the network, collecting aggregated or sampled data from the network [36, 43, 48]. Systems supporting packet-level analytics offered limited flexibility as they only supported a limited set of analytics queries [28, 33, 50, 51]. More recently, hybrid telemetry systems [19, 22, 42] struck a balance between flexibility and scale, supporting dataflow operators over packet fields at scale. Though these works enabled packet-level or flow-level network streaming analytics, they were not suited for ingesting physical, data link, and network link layer data to diagnose optical events. Previous works [34, 37] did propose a telemetry system explicitly designed for optical networks. However, they evaluated the proposed artifacts in lab environments, making it difficult to assess their performance in production settings. In contrast, OpTel demonstrates the feasibility to collect fine-grained optical telemetry data at higher frequencies (i.e., one-second granularity) by running in production at Tencent's optical backbone network for six months.

Optical layer control. Several works have studied the control interface of optical networks. Cox [11] proposed an ultimate goal of controlling the open optical line system (i.e., vendor-free optical system) in Microsoft's optical backbone by a unified SDN controller and discussed some issues surrounding the effort. Filer et al. [15] expressed a long-term goal of unifying the optical control plane and pointed out the challenges in properly controlling the plurality of optical source and line system options. They recognized Yang model [7] and SNMP [10] as potential starting points for a standard data model and control interface. In contrast to previous works which only provided the preliminary idea, we demonstrate the feasibility of a centralized control of vendor-free optical networks by designing a standardized model for devices that abstracts away the vendor-specific details.

Optical layer characterization. Previous work [9, 14, 46, 47] characterized the dispersion (e.g., polarization mode dispersion, chromatic dispersion) of the deployed fiber cable. Our work complements these efforts by investigating similar phenomenons (and more) for a much larger deployment. Ghobadi et al. [17] reported a three-month study of Q-factor data from Microsoft's optical backbone and evaluated whether fiber segments can support higher-order modulations to increase network bandwidth. The following work RADWAN [39] pro-

vided a traffic engineering system that dynamically adapted link rates according to the SNR to enhance network throughput and availability. These works took advantage of one coarsely sampled indicator. In contrast, our work benefits from the fine-grained data and a centralized controller to support inter-device analysis to detect and troubleshoot optical events. We leave correlations of IP layer performance and optical events to future work.

Diagnosis optical events. Ghobadi et al. [18] studied Q-factor data from Microsoft's optical backbone network and observed that network outages could be predicted based on the values drops in optical signal quality. RAIL [53] regarded RxPower as a key indicator of optical layer performance and found that instances of low Rx power could cause packet corruption. CorrOpt [52] used an optical layer monitor with Tx and Rx power to help determine the root cause of packet corruption in DCNs. These previous works adopted SNMP optical MIB [35] to poll optical performance indicators spanning from 5 minutes to 15 minutes. As a result, their works were slow in detecting persistent events and not capable of detecting ephemeral events. In contrast, OpTel is an optical telemetry system that supports one-second granularity optical data collection. Meanwhile, based on the signature learned from such fine-grained data, OpTel is able to detect and troubleshoot optical events in a timely manner.

6 Conclusion

This paper presents OpTel, an optical telemetry system that uses a centralized vendor-agnostic controller to collect optical data in a streaming fashion. More specifically, it offers flexible vendor-agnostic interfaces between the devices and the controller and offloads data-management tasks from the devices to the controller. As a result, OpTel enables the collection of fine-grained optical telemetry data at the one-second granularity. It has been running in Tencent's optical backbone network for the past six months. Compared to existing telemetry systems, OpTel accurately detects 2× more optical events, half of which are ephemeral events. OpTel also enables troubleshooting of these optical events in a few seconds, which is orders of magnitude faster than the state-of-the-art.

This work does not raise any ethical issues.

Acknowledgments

References

[1] grpc: a high performance, open-source universal rpc framework. https://grpc.io/.

[2] The need for otn in data center interconnect (dci) transport, 2016.

[3] Inter-datacenter bulk transfers with codedbulk. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)* (Apr. 2021), USENIX Association.

[4] ALIPOURFARD, O., MOSHREF, M., ZHOU, Y., YANG, T., AND YU, M. A comparison of performance and accuracy of measurement algorithms in software. In *Proceedings of the Symposium on SDN Research* (2018), pp. 1–14.

[5] ALIZADEH, M., GREENBERG, A., MALTZ, D. A., PADHYE, J., PATEL, P., PRABHAKAR, B., SENGUPTA, S., AND SRIDHARAN, M. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference* (2010), pp. 63–74.

[6] ARNOLD, T., HE, J., JIANG, W., CALDER, M., CUNHA, I., GIOTSAS, V., AND KATZ-BASSETT, E. Cloud provider connectivity in the flat internet. In *Proceedings of the ACM Internet Measurement Conference* (2020), pp. 230–246.

[7] BJORKLUND, M., ET AL. Yang-a data modeling language for the network configuration protocol (netconf).

[8] BOGLE, J., BHATIA, N., GHOBADI, M., MENACHE, I., BJØRNER, N., VALADARSKY, A., AND SCHAPIRA, M. Teavar: striking the right utilization-availability balance in wan traffic engineering. In *Proceedings of the ACM Special Interest Group on Data Communication*. 2019, pp. 29–43.

[9] BOHATA, J., JAROS, J., PISARIK, S., ZVANOVEC, S., AND KOMANEC, M. Long-term polarization mode dispersion evolution and accelerated aging in old optical cables. *IEEE Photonics Technology Letters 29*, 6 (2017), 519–522.

[10] CASE, J., FEDOR, M. S., SCHOFFSTALL, M. L., AND DAVIN, J. Simple network management protocol (snmp). *RFC 1098* (1989), 1–34.

[11] COX, J. Sdn control of a coherent open line system. In *Optical Fiber Communication Conference* (2015), Optical Society of America, pp. M3H–4.

[12] DI VENTRA, M., PERSHIN, Y. V., AND CHUA, L. O. Circuit elements with memory: memristors, memcapacitors, and meminductors. *Proceedings of the IEEE 97*, 10 (2009), 1717–1724.

[13] ENNS, R., BJORKLUND, M., SCHOENWAELDER, J., AND BIERMAN, A. Network configuration protocol (netconf).

[14] FEUERSTEIN, R. J. Field measurements of deployed fiber. In *National Fiber Optic Engineers Conference* (2005), Optical Society of America, p. NThC4.

[15] FILER, M., GAUDETTE, J., GHOBADI, M., MAHAJAN, R., ISSENHUTH, T., KLINKERS, B., AND COX, J. Elastic optical networking in the microsoft cloud. *Journal of Optical Communications and Networking 8*, 7 (2016), A45–A54.

[16] GENG, Y., LIU, S., YIN, Z., NAIK, A., PRABHAKAR, B., ROSENBLUM, M., AND VAHDAT, A. {SIMON}: A simple and scalable method for sensing, inference and measurement in data center networks. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)* (2019), pp. 549–564.

[17] GHOBADI, M., GAUDETTE, J., MAHAJAN, R., PHANISHAYEE, A., KLINKERS, B., AND KILPER, D. Evaluation of elastic modulation gains in microsoft's optical backbone in north america. In *2016 Optical Fiber Communications Conference and Exhibition (OFC)* (2016), IEEE, pp. 1–3.

[18] GHOBADI, M., AND MAHAJAN, R. Optical layer failures in a large backbone. In *Proceedings of the 2016 Internet Measurement Conference* (2016), pp. 461–467.

[19] GUPTA, A., HARRISON, R., CANINI, M., FEAMSTER, N., REXFORD, J., AND WILLINGER, W. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 conference of the ACM special interest group on data communication* (2018), pp. 357–371.

[20] HONG, C.-Y., KANDULA, S., MAHAJAN, R., ZHANG, M., GILL, V., NANDURI, M., AND WATTENHOFER, R. Achieving high utilization with software-driven wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (2013), pp. 15–26.

[21] HONG, C.-Y., MANDAL, S., AL-FARES, M., ZHU, M., ALIMI, R., BHAGAT, C., JAIN, S., KAIMAL, J., LIANG, S., MENDELEV, K., ET AL. B4 and after: managing hierarchy, partitioning, and asymmetry for availability and scale in google's software-defined wan. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (2018), pp. 74–87.

[22] HUANG, Q., SUN, H., LEE, P. P., BAI, W., ZHU, F., AND BAO, Y. Omnimon: Re-architecting network telemetry with resource efficiency and full accuracy. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication* (2020), pp. 404–421.

[23] JAIN, S., KUMAR, A., MANDAL, S., ONG, J., POUTIEVSKI, L., SINGH, A., VENKATA, S., WANDERER, J., ZHOU, J., ZHU, M., ET AL. B4: Experience with a globally-deployed software defined wan. *ACM SIGCOMM Computer Communication Review 43*, 4 (2013), 3–14.

[24] JIN, X., LI, Y., WEI, D., LI, S., GAO, J., XU, L., LI, G., XU, W., AND REXFORD, J. Optimizing bulk transfers with software-defined optical wan. In *Proceedings of the 2016 Conference of the ACM Special Interest Group on Data Communication* (2016), pp. 87–100.

[25] KACHRIS, C., AND TOMKOS, I. A survey on optical interconnects for data centers. *IEEE Communications Surveys & Tutorials 14*, 4 (2012), 1021–1036.

[26] LABOVITZ, C., IEKEL-JOHNSON, S., MCPHERSON, D., OBERHEIDE, J., AND JAHANIAN, F. Internet inter-domain traffic. *ACM SIGCOMM Computer Communication Review 40*, 4 (2010), 75–86.

[27] LAOUTARIS, N., SIRIVIANOS, M., YANG, X., AND RODRIGUEZ, P. Inter-datacenter bulk transfers with netstitcher. In *Proceedings of the ACM SIGCOMM 2011 Conference* (2011), pp. 74–85.

[28] LIU, Z., MANOUSIS, A., VORSANGER, G., SEKAR, V., AND BRAVERMAN, V. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference* (2016), pp. 101–114.

[29] MCQUISTIN, S., UPPU, S. P., AND FLORES, M. Taming anycast in the wild internet. In *Proceedings of the Internet Measurement Conference* (2019), pp. 165–178.

[30] MILLS, D. *RFC1305: Network Time Protocol (Version 3) Specification, Implementation*. RFC Editor, 1992.

---

[31] MIZUOCHI, T. Recent progress in forward error correction and its interplay with transmission impairments. *IEEE Journal of Selected Topics in Quantum Electronics 12*, 4 (2006), 544–554.

[32] MOSHREF, M., YU, M., GOVINDAN, R., AND VAHDAT, A. Trumpet: Timely and precise triggers in data centers. In *Proceedings of the 2016 ACM SIGCOMM Conference* (2016), pp. 129–143.

[33] NARAYANA, S., SIVARAMAN, A., NATHAN, V., GOYAL, P., ARUN, V., ALIZADEH, M., JEYAKUMAR, V., AND KIM, C. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017), pp. 85–98.

[34] PAOLUCCI, F., SGAMBELLURI, A., CUGINI, F., AND CASTOLDI, P. Network telemetry streaming services in sdn-based disaggregated optical networks. *Journal of Lightwave Technology 36*, 15 (2018), 3142–3149.

[35] PRESUHN, R., CASE, J., MCCLOGHRIE, K., ROSE, M., AND WALD-BUSSER, S. Management information base (mib) for the simple network management protocol (snmp). Tech. rep., STD 62, RFC 3418, December, 2002.

[36] RASLEY, J., STEPHENS, B., DIXON, C., ROZNER, E., FELTER, W., AGARWAL, K., CARTER, J., AND FONSECA, R. Planck: Millisecond-scale monitoring and control for commodity networks. *ACM SIG-COMM Computer Communication Review 44*, 4 (2014), 407–418.

[37] SADASIVARAO, A., JAIN, S., SYED, S., PITHEWAN, K., KANTAK, P., LU, B., AND PARASCHIS, L. High performance streaming telemetry in optical transport networks. In *Optical Fiber Communication Conference* (2018), Optical Society of America, pp. Tu3D–3.

[38] SAEED, A., GUPTA, V., GOYAL, P., SHARIF, M., PAN, R., AMMAR, M., ZEGURA, E., JANG, K., ALIZADEH, M., KABBANI, A., ET AL. Annulus: A dual congestion control loop for datacenter and wan traffic aggregates. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication* (2020), pp. 735–749.

[39] SINGH, R., GHOBADI, M., FOERSTER, K.-T., FILER, M., AND GILL, P. Radwan: rate adaptive wide area network. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (2018), pp. 547–560.

[40] STONE, J., AND PARTRIDGE, C. When the crc and tcp checksum disagree. *ACM SIGCOMM computer communication review 30*, 4 (2000), 309–319.

[41] TAMMANA, P., AGARWAL, R., AND LEE, M. Simplifying datacenter network debugging with pathdump. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)* (2016), pp. 233–248.

[42] TAMMANA, P., AGARWAL, R., AND LEE, M. Distributed network monitoring and debugging with switchpointer. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)* (2018), pp. 453–456.

[43] TILMANS, O., BÜHLER, T., POESE, I., VISSICCHIO, S., AND VAN-BEVER, L. Stroboscope: Declarative traffic mirroring on a budget. In *Proc. of NSDI* (2018).

[44] WALLS, F. L., AND VIG, J. R. Fundamental limits on the frequency stabilities of crystal oscillators. *IEEE transactions on ultrasonics, ferroelectrics, and frequency control 42*, 4 (1995), 576–589.

[45] WOHLFART, F., CHATZIS, N., DABANOGLU, C., CARLE, G., AND WILLINGER, W. Leveraging interconnections for performance: the serving infrastructure of a large cdn. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (2018), pp. 206–220.

[46] WOODWARD, S., NELSON, L., FEUER, M., ZHOU, X., MAGILL, P., FOO, S., HANSON, D., SUN, H., MOYER, M., AND O'SULLIVAN, M. Characterization of real-time pmd and chromatic dispersion monitoring in a high-pmd 46-gb/s transmission system. *IEEE Photonics Technology Letters 20*, 24 (2008), 2048–2050.

[47] WOODWARD, S. L., NELSON, L. E., SCHNEIDER, C. R., KNOX, L. A., O'SULLIVAN, M., LAPERLE, C., MOYER, M., AND FOO, S. Long-term observation of pmd and sop on installed fiber routes. *IEEE Photonics Technology Letters 26*, 3 (2013), 213–216.

[48] YU, D., ZHU, Y., ARZANI, B., FONSECA, R., ZHANG, T., DENG, K., AND YUAN, L. dshark: A general, easy to program and scalable framework for analyzing in-network packet traces. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)* (2019), pp. 207–220.

[49] ZHONG, Z., GHOBADI, M., KHADDAJ, A., LEACH, J., XIA, Y., AND ZHANG, Y. Arrow: Restoration-aware traffic engineering.

[50] ZHOU, Y., SUN, C., LIU, H. H., MIAO, R., BAI, S., LI, B., ZHENG, Z., ZHU, L., SHEN, Z., XI, Y., ET AL. Flow event telemetry on programmable data plane. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication* (2020), pp. 76–89.

[51] ZHU, Y., KANG, N., CAO, J., GREENBERG, A., LU, G., MAHAJAN, R., MALTZ, D., YUAN, L., ZHANG, M., ZHAO, B. Y., ET AL. Packet-level telemetry in large datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (2015), pp. 479–491.

[52] ZHUO, D., GHOBADI, M., MAHAJAN, R., FÖRSTER, K.-T., KRISH-NAMURTHY, A., AND ANDERSON, T. Understanding and mitigating packet corruption in data center networks. In *Proceedings of the 2017 Conference of the ACM Special Interest Group on Data Communication* (2017), pp. 362–375.

[53] ZHUO, D., GHOBADI, M., MAHAJAN, R., PHANISHAYEE, A., ZOU, X. K., GUAN, H., KRISHNAMURTHY, A., AND ANDERSON, T. {RAIL}: A case for redundant arrays of inexpensive links in data center networks. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)* (2017), pp. 561–576.

A The origins of telemetry data collected from optical device
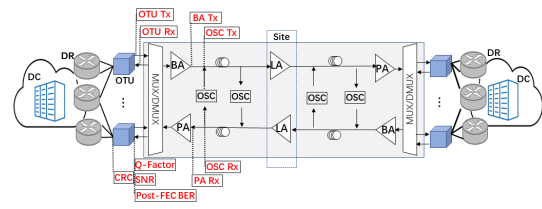
Figure 17: The origins of telemetry data collected from optical device

B Filtering out the network events from trouble tickets dataset related to optical events

Since the tickets in trouble ticket dataset describe whole network events, each ticket contains a timestamp that records the start time of the network event and the detailed alarm message and corresponding a timestamp recording the end time of the event with the causes of the failures. After manually reviewing a number of tickets, we observed that most optical events

had been saliently described in the trouble tickets. Filtering out and grouping these tickets requires a lot of effort. We design a two-layer filtration. Specifically, in the first layer, we filter out the trouble tickets related to the optical backbone network by matching keywords, phrases, and regular expressions to get a set of optical trouble tickets. In the second layer, by manually reviewing the optical trouble tickets, we observe that the optical events can be categorized into a small number of classes, i.e., fiber cable, hardware and power event. We classify these tickets based on matching keywords or phrases. In some instances, there may be multiple tickets pertaining to the same failure event. Grouping these multiple tickets into a single event requires some piece of information to be repeated in each ticket.
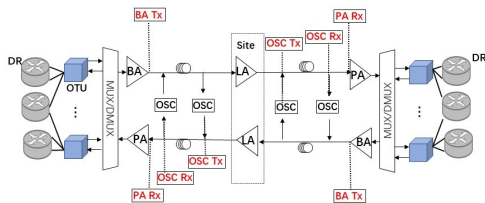
C Data collection point of power event.

Figure 18: Schematic diagram of data collection

The performance data shown in 16(a) is collected from the top part of Figure 18, and the performance data shown in 16(b) is collected from the bottom part of Figure 18. Note, the LAs in the site share the electrical power sources.

Bluebird: High-performance SDN for Bare-metal Cloud Services

Manikandan Arumugam[1], Deepak Bansal[3], Navdeep Bhatia[1], James Boerner[3], Simon Capper[1],
Changhoon Kim[2], Sarah McClure[3], Neeraj Motwani[3], Ranga Narasimhan[3], Urvish Panchal[1],
Tommaso Pimpo[3], Ariff Premji[1], Pranjal Shrivastava[3], and Rishabh Tewari[3]

Arista[1], Intel[2], Microsoft[3]

Abstract

The bare-metal cloud service is a type of IaaS (Infrastructure as a Service) that offers dedicated server hardware to customers along with access to other shared infrastructure in the cloud, including network and storage.

This paper presents our experiences in designing, implementing, and deploying Bluebird, the high-performance network virtualization system for the bare-metal cloud service on Azure. Bluebird's data plane is built using high-performance programmable switch ASICs. This design allows us to ensure the high performance, scale, and custom forwarding capabilities necessary for network virtualization on Azure. Bluebird employs a few well-established technical principles in the control plane that ensure scalability and high availability, including route caching, device abstraction, and architectural decoupling of switch-local agents from a remote controller.

The Bluebird system has been running on Azure for more than two years. During this time, it has served thousands of bare-metal tenant nodes and delivered full line-rate NIC speed of bare-metal servers of up to 100Gb/s while ensuring less than 1μs of maximum latency at each Bluebird-enabled SDN switch. We share our experiences of running bare-metal services on Azure, along with the P4 data plane program used in the Bluebird-enabled switches.

1 Introduction

For some time now, Software Defined Networks (SDNs) have been foundational in enabling virtualized networks for customer workloads in multi-tenant clouds. Traditionally, the data plane of SDNs has been implemented in software as part of the end-host networking stack, typically leveraging virtual switches in hypervisors such as Open V-Switch (OVS) [16] or user-level networking libraries such as DPDK [17]. Given that scale and performance needs have grown over the years, the mechanisms available to offload such software-based packet processing have also evolved. These include solutions such as smartNICs [15], which leverage Switch-on-a-Chip (SoCs), ASICs, and FPGAs to perform packet processing at line rate without incurring significant overhead [58–60].

Today, cloud customers have even more demanding workloads in the cloud as they look to migrate their line-of-business applications and begin to phase out their own data centers.

These workloads require complete control of the hardware, and in many cases, custom hardware to be hosted in the cloud. For example, workloads such as those for NetApp, Cray, SAP, and HPC [13, 53] require custom hardware. We refer to the cloud offering for supporting such workloads as bare-metal cloud services or hardware as a service (HWaaS).

Bare-metal workloads are not well-supported by traditional SDN stack implementations. In general, bare-metal servers do not offer the necessary opportunities for integration with the networking stack on the host or NIC, calling instead for a "bump-in-the-wire" approach that has no dependency on the host hardware. Since the Top-of-Rack (ToR) switches are the first network hop connected to these hosts, the ToR offers an excellent opportunity to implement this "bump in the wire."

In this paper, we introduce Bluebird, a ToR-based SDN solution that is broadly deployed in one of the largest public cloud infrastructures to enable bare-metal workloads. We also discuss the challenges, design, and operational experiences in building and designing such a solution.

Bluebird is based on programmable ASICs such as the now largely available Barefoot Tofino chipset [27] as well as upcoming merchant silicon offerings like Broadcom's SmartTOR ASIC [6]. The programmability of high-speed networking ASICs, along with the increase in scale, have made the ToR-based "bump-in-the-wire" approach feasible. Cloud providers must be able to evolve the SDN capabilities of the platform as customer requirements change. Without programmable chips, a cloud provider may have to wait for an 18-24 month technology cycle before changing their service offerings. Unless, of course, the cloud provider undertakes an off-cycle, expensive hardware replacement. Additionally, several SDN functions, such as load balancing, NAT, etc., require flow state tracking. ToR ASICs such as Barefoot Network's Tofino and Broadcom's SmartToR are now able to track millions of flows, which is critical to enable network virtualization in a ToR.

Bluebird is able to achieve line-rate throughput and deliver latencies of less than 1μs that are on par with non-virtualized environments. By leveraging route caching mechanisms, Bluebird can scale to the largest virtualized networks that exist in a public cloud. The rest of this paper is organized as follows: §2 reviews different SDN implementations; §3 describes goals and rationales behind Bluebird; §4 presents the design at the
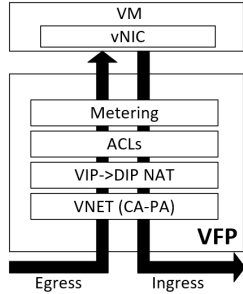
Figure 1: Virtual Filtering Platform (VFP) design.

base of our solution; §5 investigates performance; §6 explores operationalization and experience; §7 discusses related work; and §8 concludes this paper.

2 Background

In this section, we present an overview of the SDN stack implementations coexisting within Azure and how they compare with Bluebird to enable bare-metal services. Table 1 summarizes the comparison.

2.1 Host SDN

First, we consider end-host based software solutions. Figure 1 shows the Host SDN model, where the SDN software stack runs on the host machine hypervisor. Packets to and from the VMs are processed by a programmable virtual switch (vSwitch) operating within the hypervisor environment. The vSwitch's design is critical to the effectiveness of this solution since it is responsible for implementing the forwarding policy while still minimizing overhead. Since the host has already processed every packet reaching the physical forwarding layer, the physical networking equipment can be relatively simple. However, processing packets in software is expensive and competes with client software for host resources. This results in reduced revenue and has a negative impact on network efficiency as congestion increases CPU use [15].

Using Hyper-V as the hypervisor and the Virtual Filtering Platform (VFP) [14] as the vSwitch, Azure primarily employs the Host SDN model across its fleet (Figure 1). The VFP implements SDN policies by acting as a programmable platform accessible to the controllers running Azure's SDN. The VFP is organized in layers, which are stateful flow tables that enforce the controller's policy. Each layer implements a specific set of inbound and outbound rules that can filter and transform packets. A packet traverses the layers in order, matching one rule per layer by searching by rule priority.

Figure 1 shows a common layer configuration. Following the inbound order, the Virtual Network (VNET) layer provides tunneling for packets coming from Customer Address (CA) space to the Physical Address space (PA). Inbound packets are decapsulated while outbound packets are encapsulated.

The next layer is the Ananta [61] NAT load balancer layer, which NATs inbound packets from a Virtual IP (VIP) to a Direct IP (DIP). The Access Control Lists (ACLs) layer is a stateful firewall, while the metering layer is for billing and is placed as the last layer between the VFP and the VM.

While the Host SDN model is used widely in Azure, it is not well-suited for bare-metal workloads. Such workloads are not Hyper-V based, leaving no environment to implement the VFP. Furthermore, the network performance is expected to be on par with deployments in non-virtualized environments, precluding a software-based solution.

2.2 SmartNIC-based SDN

A smartNIC is a programmable network interface card (NIC) that supports the network processing operations usually performed by the host CPU. SmartNICs can be configured for both control plane and data plane operations. They can make use of various technologies depending on the requirements: Application-specific Integrated Circuit (ASIC), System-on-chip (SoC), and Field-programmable Gate Arrays (FPGA).

Compared to VFPs, smartNICs offer lower latency and higher throughput while maintaining the same scalability and programmability level. This allows the host CPU to offload some costly network operations, resulting in reduced processing time and improved power efficiency. However, in some cases, only a subset of tasks is offloaded to the smartNIC, which in itself requires careful orchestration between the hypervisor and the smartNIC operating system.

While smartNICs have proven to be effective overall, they are not a good fit for the bare-metal model given the integration requirements. This is mostly due to the complexities that would be introduced at the hypervisor and network stack of the host. Additionally, the SDN stack implementation should be decoupled from any particular or specialized bare-metal appliance to ensure that a single, general approach will be compatible with all bare-metal services.

2.3 SDN on ToR

Traditionally, data centers are built using fixed-function switches. In such environments, the cloud intelligence resides in the host or hypervisor and not in the network. The host uses smartNICs and VFPs to define a clear separation between the control and data plane.

SDN on ToR refers to the use of programmable switches to execute policy on the ToR switch instead of the host. In order to use a programmable ToR, the target function(s) must be well-defined and limited in scope. A well-targeted SDN ToR application can reduce development time and complexity.

Most, if not all, SDN policies could be supported by programmable switches. However, we have taken an incremental approach where only a small subset of features is initially introduced. This set will be expanded once it has met certain

	End-host software stack per core (§2.1)	SoC-based smart-NIC (§2.2)	ASIC-based smart-NIC (§2.2)	FPGA-based smart-NIC (§2.2)	Programmable ASIC-based ToR (Bluebird) (§2.3)
Max Throughput	< 40Gbps & 10's of Mpps	Up to ~100Gbps & ~100Mpps	Up to 200Gbps & 100-200Mpps	Up to 200Gbps & 100-200Mpps	6.4-12.8Tbps & 5-7Bpps
Latency	< 100 μsec	> 1 μsec	> 1 μsec	> 1 μsec	< 1 μsec
Scale	GBs of DRAM + traditional cache hierarchy	8 GBs of DRAM + traditional cache hierarchy	Tens of MBs on chip cache + GBs of DRAM	Tens of MBs on chip BRAM + GBs of DRAM	12 stages of high-throughput pipeline and each pipeline has 24 TCAMs and 80 SRAM blocks
Cost per 100Gbps	Medium (including capex and opportunity cost)	Medium	Medium	Medium	Low
Power consumption per 100Gbps	~500-700W per server including the NIC	~500-700W per server including the NIC	~500-700W per server including the NIC	~500-700W per server including the NIC	~300W for the system that includes 64 ports of 100GbE (~5W/100GbE)

Table 1: Comparison of SDN stack implementations.

standards of quality and reliability. In Bluebird, the primary objective is for the ToR to maintain a high number of CA-to-PA mappings while ensuring hardware-like performance.

An SDN ToR gives us the ability to collapse multiple functions into one network element. In the Bluebird model, we collapse two key roles into a single device: 1) logical network isolation between customers via Virtual Routing and Forwarding (VRF) instances and 2) the association of one or more CA-to-PA mappings per VRF per customer. Implementing these two functions separately on different devices (such as routers implementing VRFs with tunnels to servers running software gateways) incurs the cost of routers and additional servers. By collapsing the routing and CA-to-PA mapping tasks onto a single device, we reduce the number of hops, encapsulations, and consequently latency for bare-metal workloads. Implementing these functionalities directly on the SDN switch results in increased performance and scalability.

To implement an SDN ToR for use with Bluebird, a single VRF is allocated per customer to guarantee logical isolation between customers. Since the goal of the SDN ToR is to connect a customer's bare-metal instance to their VNET, each VRF is programmed with CA-to-PA mappings in the form of VXLAN [48] static routes that associate the bare-metal host to its VNET. Customized P4 programming is used to perform the necessary encapsulation for packets destined to the VNET. This allows the communication between bare-metal (BM) and VMs and between BM and BM to happen at hardware speeds. Additionally, the number of programmable routes is extended through an onboard cache that increases the otherwise limited number of routes the switch ASIC on-chip memory can hold. The route-cache solution is discussed in more detail in §4.

2.4 SDN Servers

The bump-in-the-wire method could have also been approached by assigning dedicated ports on a custom DPDK-enabled SDN server attached to the bare-metal appliance, making use of DPDK-enabled smartNICs on this server to perform the bump-in-the-wire function. However, based on the power consumption data in the Table 1, one can deduce that dedicated servers that carry out bump-in-the-wire operations make the power overhead a non-starter. The performance and scale that an SDN ToR offers in a <500W power envelope makes a strong case for using a dedicated SDN ToR.

3 Design Goals and Rationale

In developing an SDN solution for bare-metal workloads in Azure, we had the following objectives:

1. *Programmability*
   SDN for bare-metal workloads needs to be able to evolve along with the rest of the SDN stack. The VFP [14] model enables many configurable virtual network features. As requirements and policies change over time, SDN for bare-metal should maintain interoperability with the existing stack. This is achieved through the ToRs' programmability which provides control at every stage of packet processing.

2. *Scalability*
   The most significant disadvantage of SDN on ToR compared to host implementations is the limited scale. Memory linearly scales with the number of hosts. Consequently, route capacity can quickly become a bottleneck in resource-limited ToRs. Accordingly, we developed a cache system that extends the hardware capacity of our ToRs and allows us to meet our scalability and performance requirements.

3. *Latency and throughput*
   Bare-metal workloads typically demand high bandwidth, low latency, and deterministic behavior. To meet these requirements, we have used programmable high-speed network ASICs since they offer consistent latency, high throughput, and sustained performance.

4. *High availability*
   To avoid customer impact due to hardware failure or maintenance, the SDN for bare-metal solutions must have high

availability. To support this requirement, we designed redundancy into Bluebird as described in §6.

5. *Multitenancy support*

Azure supports a large number of customers and tenants who have the ability to create, modify, and delete virtual networks rapidly. When supporting multitenancy, isolation is critical for providing an experience indistinguishable from dedicated networks and servers.

6. *Minimal overhead on host resources*

With the VFP model, VMs running on the host compete with the SDN stack for hardware resources. The introduction of AccelNet [15] and FPGA-based smartNICs has significantly reduced the overhead, but the VFP still stays on the host to process the first packet in the flow. With bare-metal workloads, customers expect the performance to be similar to that of direct access to the underlying hardware.

7. *Seamless integration*

Bare-metal workloads run on a wide array of architectures and operating systems. Integrating a new workload in the Azure network should be possible without change to the bare-metal server. Bluebird decouples the workload architecture from the SDN stack and enables consistent virtualization of a diverse set of bare-metal workloads.

8. *External network access*

Given that bare-metal hosts may require Internet or external network access, a form of Network Address Translation, directly available on the SDN ToR, should be supported.

9. *Interoperability*

As we introduce programmable ToRs to support bare-metal workloads, these ToRs need to transparently operate with the existing SDN stack to ensure communication between the physical and virtual address space. Interaction with the VFP §4 is of primary importance to realize a heterogeneous system like the one proposed in this paper.

4   System Design

In network system design, there is often a trade-off between the cost of a device, its memory (or route capacity), and features intrinsic to the Network Processing Unit (NPU) or ASIC itself. Internet core routers are generally feature-rich, designed to support large route tables, able to move large amounts of network traffic (>30 Tbps), and usually quite expensive. Alternatively, the ToRs in data centers are cost-effective but have fewer features and support smaller routing tables. In our design, we needed an SDN ToR with a reasonably large VNET routing table but with the cost efficiencies of a typical data-center-class ToR. Bare-metal hosts would connect directly to such SDN ToRs and use a specialized route table to communicate with Azure VMs in a virtualized address space.

Before Bluebird was introduced, Azure supported on-prem bare-metal to VNET connectivity through a software-gateway model. In this model, traffic is encapsulated on a router and

forwarded to one or more software-gateways. The software gateways, implemented on standard servers, hold large numbers of CA-to-PA mappings associating an on-prem customer to their VNET. However, with the introduction of the workloads for NetApp, it became clear that the gateway model would not meet the throughput and performance requirements. The NetApp bare-metal service required at least 240Gbps of throughput with a latency ceiling of no more than 4ms. With this in mind, we decided to adopt the SDN ToR model and program the CA-to-PA VNET routes directly onto the ToR, avoiding the software gateway altogether. This improves the throughput as it is now limited only by the throughput of the SDN ToR, which in Bluebird's case is 6.4Tbps.

A considerable effort was put into the planning of how on-chip resources had to be arranged in the pipeline to meet our needs. In particular, we had to decide how to allocate the on-chip switch memory to maximize the number of CA-to-PA mappings which are represented in the VXLAN Tunnel Endpoints (VTEP) resource table. Using Tofino's P4 programmable pipeline, we reduced the IPv4 and IPv6 unicast route table size and significantly increased the VXLAN VTEP table scale from 16K to 192K entries. The ability to support 192K CA-to-PA mappings offered greater flexibility in customer address choices since many specific routes (/32 IPv4 or /128 IPv6 routes) could be used to point to customer VNETs rather than limiting to a smaller number of aggregate routes. In order to make the design future proof, we gave ourselves the maximum allowable table space on the chip in the event that more specific routes were dominant.

The reduction in IPv4 and IPv6 table space also allowed us to extend the VXLAN Network Identifier field space as well as add support for an IPv6 underlay. The custom P4 programmability proved to be extremely valuable in helping us achieve our scale objectives.

While the P4 profile we implemented to give the SDN ToR a large VXLAN VTEP table was adequate when we launched the service, we had to start planning for growth. The VXLAN VTEP route capacity of the ToR was further enhanced with the introduction of a route cache system. The route cache mechanism allowed the VXLAN VTEP table capacity to grow beyond the hardware limit of 192K entries. The caching solution is described later in this section (§4.4). P4 programming flexibility also allowed for other quick packet header manipulations that helped in the rapid development of this service, namely overwriting the inner Ethernet header with the destination VM's MAC address while modifying the VXLAN UDP source port to a custom value.

In the remainder of this section, we discuss the packet flow, related packet transformations, and the control plane design.

4.1   Packet Flow

In this section we provide an in-depth discussion on the packet flow. In order to achieve customer isolation at a logical
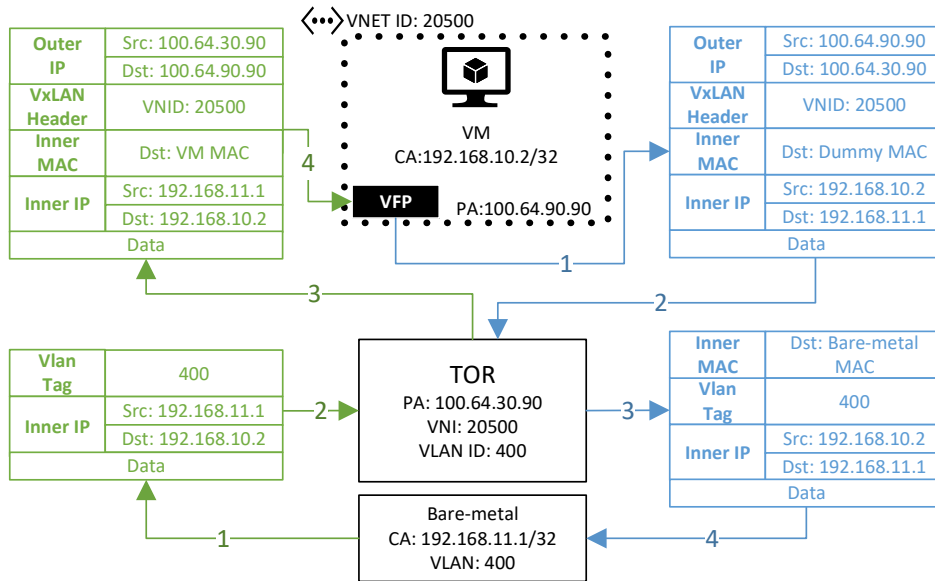
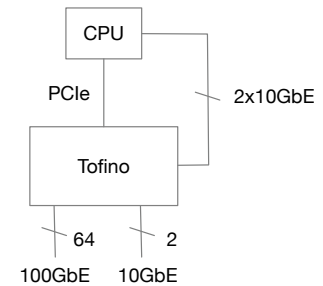Figure 2: Packet flow between a VM and a bare-metal server.

Figure 3: Front panel and CPU interfaces in an Arista 7170.

layer within a common fabric, we identify each customer's VNET by a Virtual Network Identifier (VNI) associated with a unique Virtual Routing and Forwarding (VRF) instance.

Bare-metal to VM. When a bare-metal server sends a packet to an Azure VM, a VLAN tag is added to the outbound packet. The ToR then receives the packet on the virtual interface specified by the VLAN tag. Each interface on the ToR has an associated VRF configured to route the packet to the customer VM. The routes in the VRF associate a VM's IP with the routable IP of the host containing the VM, the VNI of the VNET, and the MAC address of the VM. When the packet reaches the destination host, the VFP decapsulates the packet and uses the MAC to switch the packet to the correct VM. This flow is explained with an example below.

In the green flow shown in Figure 2 at point (1), the bare-metal server sends a packet to the VM through the VLAN 400 interface. The packet is then received on the associated interface on the server's ToR at (2). On the ToR, VLAN 400 is configured to be associated with the VRF 20500, which contains the routes programmed by Bluebird to route the packet to the VM. At (3), the ToR rewrites the inner destination MAC with the VM's MAC contained in the VRF. At (4), the ToR, configured to use the loopback interface as the VXLAN source interface, encapsulates the original frame in a VXLAN frame containing its own PA 100.64.30.90 (loopback IP address) as outer source IP and the VM's host PA 100.64.90.90 as the outer destination IP.

VM to Bare-metal. When an Azure VM sends a packet to a bare-metal server, a Bluebird-provisioned rule instructs the VFP on the host to encapsulate the Ethernet frame in a UDP datagram containing the customer VNI and the IP of the bare-metal's ToR as the destination VTEP IP. The SDN ToR decapsulates the packet and uses the VNI to identify the VRF.

Within the VRF, a route lookup is performed to identify the next-hop to which the packet is subsequently be forwarded.

In the blue flow in Figure 2, a VM sends a packet to the bare-metal server where the inner source and destination IPs are respectively set to the VM's CA 192.168.10.2 and the server's CA 192.168.11.1. The destination MAC is set to a dummy value. In the VFP (1) the packet is encapsulated in a VXLAN frame containing the VM's VNI 20500, the outer source IP pointing to the host PA 100.64.90.90, and the PA 100.64.30.90 of bare-metal's ToR as destination IP. At (2), the bare-metal's ToR receives the packet and decapsulates it. The virtual network identifier is used to find the VRF associated with the customer virtual network. At (3), the switch learns the destination MAC through ARP and adds a VLAN tag pointing to the configured VLAN interface 400, and at (4), the packet is routed to the bare-metal server.

4.2 Platform Selection

Using a switch ASIC with a programmable P4 pipeline, we were able to quickly prototype packet formats that would interoperate with Azure's VFP. Additionally, based on the initial requirements, we needed support for at least 192K CA-to-PA mappings. A variety of silicon offerings could meet most of our requirements at the time, but the scale of CA-to-PA mappings pointed our investigation towards (Intel) Barefoot Network's Tofino-1 chipset.

The Tofino-1 is a 6.4Tbps single-chip solution with 12 programmable stages, 256x25/10G SerDes, and a software-defined P4 packet processing pipeline. On the Arista 7170 switch, the Tofino-1 is coupled with a Quad-core 2.2GHz Intel Pentium CPU with additional 2x10GbE ports from the switch ASIC wired directly into the CPU (as shown in Figure 3).
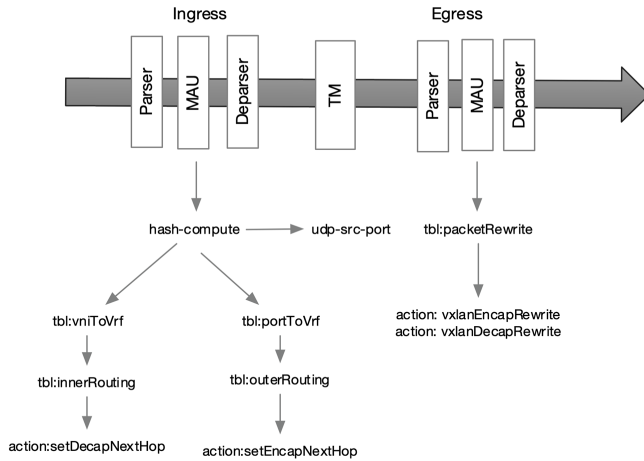
Figure 4: P4 programming pipeline.

These additional ASIC-to-CPU 10GbE ports provide a fast path for route-cached packets to be processed on the CPU.

4.3 P4 Pipeline Design

P4 facilitated rapid prototyping and quick iteration in finding a balance between desirable features and the available chip resources. For instance, while the P4 programming needed to create CA-PA mappings was easy to define, special considerations had to be given to whether the underlay would be IPv4 or IPv6. In a simplistic model, having an IPv6 underlay would significantly reduce the number of CA-to-PA mappings since the entries in the forwarding table would need additional resources to support IPv6 PA destinations. However, with our custom P4 pipeline, we were able to completely decouple the underlay route scale from the overlay route scale thus giving us the maximum number of CA-to-PA mappings independent of whether the underlay used is IPv4 or IPv6.

To describe the packet transformations used in our flows, we used the P4 programmable pipeline on Tofino, which consists of an ingress and egress block (see Figure 4), with each block comprising a sequence of sub-blocks: Parser, Match-Action-Unit (MAU), and Deparser. The Parser block parses the packet and extracts the relevant headers. After that, the MAU performs table lookups and manipulates the packet. The Deparser then reassembles and sends the packet to the Traffic Manager (TM). Finally, packet queuing, replication, and scheduling are done by the TM. A part of the P4 program [1], which illustrates one of the inner-MAC rewrite transformations implemented in our pipeline, is open-sourced.

In the sample P4 program shared [1], the parser excludes regular IP packets and keeps the VXLAN encapsulated packets. After each packet is decapsulated, a route lookup is done on the inner destination IP which determines the action to be taken and the data to be rewritten. In the egress logic, several fields, such as the inner-MAC address, are rewritten based on the bridged metadata received from the ingress pipeline.

```
ip route vrf VNET-A
192.168.10.2/32 vtep 10.100.2.4 vni 20500
router-mac-address 00:12:23:54:A2:9F
```

Figure 5: Static VXLAN route configuration on ToR

While the P4 example provided gives the reader a high-level view of the flexibility available in packet manipulation, the actual P4 code used for this service is more intricate and optimized, allowing for 192K mappings and additional features.

The flexibility that P4 provides allowed us to give the ToR a 'personality' based on the application. A ToR can be preloaded with a custom P4 profile, making the ToR suitable for a given application. For example, we use a 'bare-metal' profile when the ToR is used for Bluebird workloads and a 'NAT-profile' when source NAT is required. Each profile results in a different P4 program getting activated in hardware.

The rest of the packet transformations are presented in detail in the remainder of this section.

Inner Destination MAC Rewrite. When a BM sends traffic to a VM, the receiving hypervisor's VFP uses the inner destination MAC (DMAC) to forward the packet to the destination VM. If the DMAC is unknown, the VFP drops the packet. For this reason, the CA-to-PA routes have the form of static routes extended with additional fields. An extended route, defined on the SDN ToR, contains the VTEP as the PA address, the VNI for the VNET, and the destination MAC of the VM. When a packet matches a route, the ToR overwrites the DMAC with the MAC of the VM. For example, the route in Figure 5 points to a VNET VM with MAC 00:12:23:45:A2:9F at address 192.168.10.2 residing on a VXLAN with VNI 20500 and reachable host (PA) at 10.100.2.4.

Limiting the Range of the VXLAN Source Port. In a traditional VXLAN, the ToR imposes a VXLAN UDP source port value derived from the incoming packet's entropy. The source port is calculated per packet. This is done to help with hash-based ECMP load-balancing schemes employed by network chipsets, ensuring that VXLAN packets are properly load-balanced across the network. To aid the VFP on the VM host in identifying BM-sourced VXLAN packets, we limit the range of source port values usable by the SDN ToR. Limits in imposable ports can be set with a simple CLI command.

Inner Destination MAC Masking. We also needed customization on the SDN ToR to ignore the inner destination MAC address arriving in the VXLAN packet. This is in the direction of the VNET to the SDN ToR, where the inner destination MAC address is usually resolved by an ARP exchange between two VXLAN hosts. Specifically, an ARP request/reply exchange would have to take place, ensuring that the end-hosts are aware of each other's MAC addresses. The entire ARP resolution step can be skipped if the SDN ToR absorbs all packets regardless of the DMAC value, as described in packet flow in §4.1. The VFP transmitter on the VNET simply writes a bogus inner DMAC value in the

packet destined to the SDN ToR. The SDN ToR is made to ignore this bogus MAC and proceeds to route all the received frames regardless. The SDN ToR accomplishes this task using a wildcard bitmask applied using a CLI command.

4.4 Route Cache

Although we were able to make room for additional mappings using a custom P4 pipeline, we were faced with the challenge of increasing the scale beyond what the chip hardware could support. At this point, we had a working P4 pipeline that was programmed to fit 192K CA-PA mappings, support an IPv6 underlay, and offer 1:1 static NAT to BM hosts.

As the number of bare-metal customers grew, we realized that the 192K CA-PA upper-limit of the Tofino would soon become a bottleneck. We considered a few alternatives, one of which was to quickly onboard the next-generation Tofino (Tofino-2), which was known to support up to 1.5M CA-PA mappings in hardware. However, we ended up pursuing the route cache feature since it would be valuable regardless of the scale of the underlying Tofino ASIC. Route caching gives us a five-fold increase over the original 192K mappings. The route cache feature is an implementation of the familiar statistical multiplexing model whereby a significantly higher number of customers can share a finite resource, as long as not all customers are active at the same time. In other words, if we know that our customers are not always using all available hardware entries, we can reassign those unused entries to other active users. A primary enabler for the route cache concept is the way Bluebird provisions SDN entries. Bluebird does not preconfigure the SDN ToR but instead dynamically provisions mappings as needed allowing the route cache logic to continuously determine which entries are to remain in hardware or moved to software.

Before describing the route cache feature, it is important first to understand the Software Forwarding Engine (SFE). The SFE is a DPDK-enabled packet processing function provided by the ToR's CPU which has a packet forwarding rate of 200K pps. CPU bound packets use the 2x10GbE interfaces connecting the Tofino to the CPU to reach the SFE. A software agent on the ToR monitors hardware entries on the switch ASIC and moves inactive entries to the SFE. SFE resources are CPU and memory bound. With 16GB of memory, up to 600K mappings can be stored in the SFE, whereas 1.5M mappings can be stored with at least 32GB of memory. The number of mappings compared to the total memory is low because the memory also supports the switch operating system, i.e., EOS (Extensible Operating System). The portion of memory used by the SFE is relatively small: about 3GB out of the available 32GB total memory is used to store mappings. The rest of the memory is used for running the operating system and other agents such as the routing agent, platform agent, etc.

Bluebird configures static VXLAN routes/mappings and

Threshold level	Utilization	Idle time
low	85%	1100s
medium	90%	300s
high	95%	100s

Table 2: Default caching thresholds.

specifies whether the route/mapping is cacheable or not. The mapping itself is programmed by Bluebird using a JSON RPC call as described later in this section. A mapping is identified as active if there was a packet that recently used the prefix programmed in the VNET route. We will use the concept of a 'hitbit' to note when a route entry is touched by a packet, either in software or hardware. EOS then maintains this hitbit for all hardware and software (SFE) entries. The hardware hitbit is triggered when mappings are used for hardware-based forwarding. A SFE hitbit is triggered when a packet takes the software DPDK path, indicating that this software mapping now needs to be upgraded from the SFE to hardware.

To select the mappings that are evicted from the hardware, we use a Least Recently Used (LRU) eviction algorithm. LRU entries are found by polling the hitbit property maintained in the hardware for each prefix. The flows going through the SDN ToR are monitored using an age-based idle timer. No packet state is maintained, nor are the packets examined. We considered other hardware eviction algorithms but ultimately rejected them. These included 1) tracking TCP flows and 2) tracking flows that carried the most traffic volume.

These options were rejected because tracking TCP flows is expensive from two perspectives: 1) the need to send packets containing TCP flags S, F, R to an agent on the switch for tracking purposes, and 2) flows are expensive to store as the access key would comprise of the source IP/port and destination IP/port. For routing purposes, the switch only needs the destination IP and the added flow state becomes unnecessary overhead.

We decided to implement the idle-timer based approach since it was simpler and it met our requirements. The idle timer based approach is the simplest and most efficient because it does not require tracking individual flows or state. In the future, other algorithms may be considered as we learn more about customer traffic patterns.

In the CLI, we can specify CA-PA mapping entries as cache candidates. All candidate mappings become a part of the route cache, which means that these prefixes can be downgraded to software if they become inactive and can be upgraded to hardware if they become active. The aging time of hardware routes and how many of these entries remain in hardware can be configured as a percentage of total hardware capacity. Finally, the default threshold values are listed in Table 2.
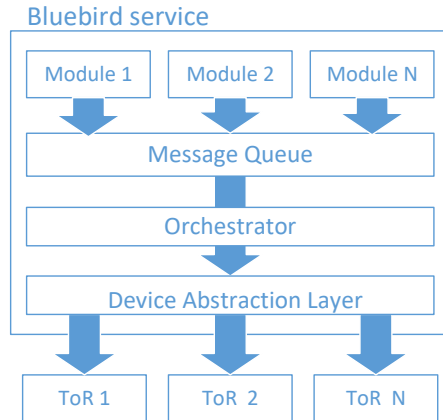
Figure 6: Bluebird service structure.

4.5 Control Plane and Policy Provisioning

In the Azure host SDN, a controller running on each host programs the VFP. However, since this is no longer an option for the bare-metal workloads, two alternatives for hosting SDN logic were considered; an agent on the ToR or programming the ToR via an external service. We rejected the idea of having an agent on the ToR since it did not meet our performance goals. The agent would compete for resources with latency-sensitive operations running on the ToR, and resource consumption could increase as requirements changed over time. On the other hand, an external service does not need immediate proximity to the ToR because configuration on a ToR has less stringent latency requirements than typical data plane operations. Moreover, a separate service also has advantages in terms of fault tolerance and deployment time. As a result, we introduced the Bluebird Service (BBS) (Figure 6).

Provisioning SDN Policy on ToR

BBS is a lightweight, multi-tenant, stateless microservice that configures the ToRs with virtual network policies for each customer (Figure 6). Each policy is represented as a JSON object that specifies a ToR's desired state, called the goal-state. Azure SDN services send their goal-states to BBS' message queue leading to the orchestrator module which parses and consolidates the goal-states into ToR configurations. Configurations are then transferred to the device abstraction layer (DAL), which keeps the SDN business logic independent from ToR implementations. In the DAL, they undergo a conversion process that results in a sequence of commands for the targeted ToR. The list of allowed commands is strictly limited to the operations needed for bare-metal provisioning, reducing the possibility of interference with other automation systems responsible for software upgrading or traffic shifting. In the case of the Arista 7170, BBS uses the JSON-RPC 2.0 protocol over HTTPS. Each JSON payload contains an ordered list of commands using Arista EOS CLI syntax.

BBS performs a sync-check on all ToR configurations at defined intervals. At every sync, it calculates the delta between a ToR's configuration and its target configuration and performs a reconciliation in case of differences. Each configuration request is atomic, and configurations are versioned to avoid inconsistent states due to out-of-order execution. BBS also ensures state consistency between multiple ToRs if they are part of a high availability network configuration (see §6.1) in which they are seen as one logical entity.

To prevent resource exhaustion due to extremely large VNETs generating a high number of routes, BBS limits the number of programmable routes per VNET. The default limit is maintained as a function of the ToR's capacity. When more routes are needed, the limit can be raised to match the requirements and, in some cases, customers are migrated to dedicated ToRs.

Azure regions protect from failures through increased redundancy. Each region is subdivided into several distinct physical locations called availability zones (AZs). Each zone is made up of one or more data centers (DCs) equipped with independent power, cooling, and networking. BBS is deployed per AZ within an Azure Service Fabric [33] ring organized as a series of active and inactive instances. Additionally, BBS's scope is not limited to a single AZ and can target any AZ within the same region. However, this is limited only to scenarios in which another AZ is severely impacted and the local BBS is unable to function.

5 Performance

Over the past two years, Azure has been deploying bare-metal services on SDN-ToRs in over 42 data centers. In addition, Azure has successfully onboarded several HWaaS native applications such as Cray ClusterStor, and NetApp Files [13,53]. As of today, we have powered several thousands of bare-metal servers and serve thousands of terabytes of traffic per day. Although a significant number of customers are adopting these bare-metal services, the number of routes tied to these workloads has yet to grow to the point of exceeding the route cache threshold of 85% capacity which would trigger the use of route cache. We estimate that the threshold will be exceeded within a year and believe that the route cache feature will play an important role in the future of the bare-metal service offering as the number of provisioned VNET routes outpaces the growth in hardware capacity of SDN ToRs.

We have compared the performance of bare-metal servers with VMs using Azure accelerated networking, both running in an Azure data center on Intel Xeon E5-2673 v4 (Broadwell at 2.3 GHz) CPUs with 40Gbps NICs and Windows Server 2019. We measured throughput, CPU overhead, and latency, with both solutions performing similarly and with no appreciable difference.

This section presents a performance analysis specifically focusing on route cache that is carried out through the use of synthetic testing tools and production data where available.
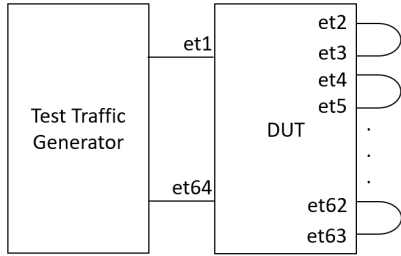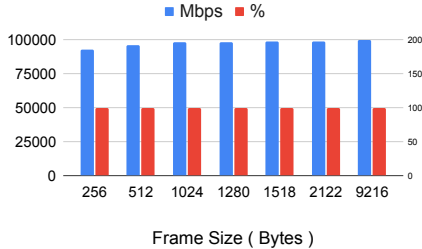
Figure 7: Test topology.

Figure 8: Throughput in Mbps on the left y-axis and in % on the right y-axis per frame size.
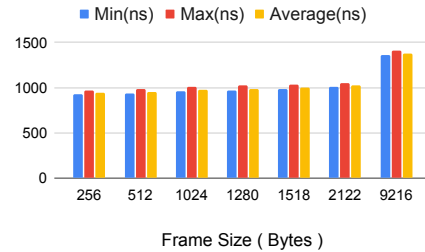

Figure 9: Latency measurements in nanoseconds (y-axis) per frame size (x-axis).
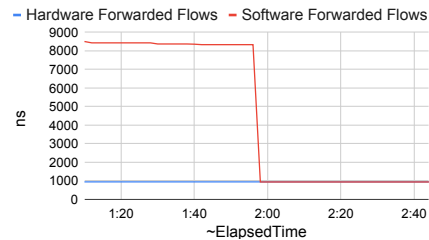

Figure 10: Latency comparison between software forwarded and hardware forwarded flows in nanoseconds per frame.

5.1 Hardware Performance

To measure the port-to-port latency within a SDN ToR, we connected a traffic generator directly to the Device Under Test (DUT). Figure 8 shows that the Intel Tofino ASIC performs at wire-speed with a consistent port-to-port latency of <1μs. The Tofino chip demonstrated deterministic performance across all cases of packet manipulation required for Bluebird. Additionally, even during heavy traffic load, minimal differences were observed between the min and max throughput values. Goodput was tested by passing L2 frames of various size through a DUT in a snake topology (Figure 7).

In Figure 8, 100% goodput is sustained across all the tested packet sizes up to 9216 bytes, with the only expected exception for packets of 256 bytes. This is due to the smaller relative size difference between header and payload. This performance is in line with our requirements to support bare-metal workloads that are bandwidth and latency-sensitive.

From a power utilization perspective, although the ToR behaves like a bump in the wire, it is not adding any power draw over a regular data center design. The typical/max power draw of BlueBird ToRs is 271W/571W. This is in the same range as other ToRs used in Azure with the same bandwidth (64x100G) which have a typical/max power draw of 314W/616W.

5.2 Performance Impact of Route Caching

The route cache feature is responsible for moving route entries from the SFE to the hardware and vice versa while ensuring the process is transparent to the customer. The CPU on the ToR provides a DPDK-enabled packet processing function, helping to meet our stringent latency requirements. As men-

tioned earlier, we use 2x10G high-bandwidth links between the CPU and the ASIC. Since the traffic consists exclusively of TCP flows, our system has enough time to move the entries from the SFE to hardware by the time the TCP 3-way handshake is complete.

There are two primary factors that contribute to the observed latency when route-caching is performed; 1) the latency experienced while packets are being forwarded in software by the SFE and 2) the time taken to move a route entry from the SFE to the hardware.

To accurately measure the latency experienced by packets forwarded in software, we used instrumented packets that were generated by a traffic analyzer and forcefully forwarded them via the SFE. This was done by defining 5000 routes on the SDN ToR and sending traffic to each one of these entries while deliberately preventing the entries from being programmed into the hardware ASIC. To ensure that these software entries would not get programmed in the hardware, we temporarily disabled the route cache feature after the route entries were activated in the SFE. This approach ensured that the route entries in the SFE remained in software while we recorded latency measurements. In this state, where the routes were present only in the SFE, we found that packets experienced an increased latency of about 8μs when compared to the latency experienced by packets using hardware entries.

Under normal circumstances, and assuming the route-entry used is in the SFE, we expect the packets to be software-routed for only a short time. The first packet that triggers a hitbit recording immediately kicks off the hardware programming for the SFE route. While this transition is hard to measure using generic traffic analyzers due to their lack of precision, we were able to inspect system logs on the SDN ToR to
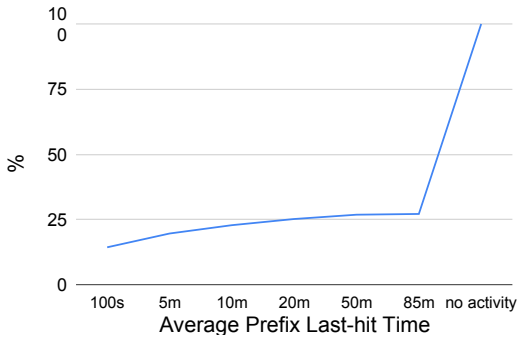
Figure 11: Prefix utilization based on last-hit time.

learn that this transition time is <2ms. To measure this, we compared the logged timestamps between when the packet hit the SFE and when the route appeared in hardware.

5.3 Validating Route Cache

The premise of designing and implementing the route cache model was based on a theoretical assumption that not all bare-metal customers would require hardware table entries at the same time. In order to prove that route caching would work, we had to find a way to record the age, or the last-time a given hardware entry was used. In other words, if an entry had aged and it's recorded "last time used" was old, then that entry could be demoted and moved to the SFE. Furthermore, we created the following time-buckets for age categories; 0s-100s, 101s-5m, 5m-10m, 10m-20m, 20m-50m, 50m-85m and a 'no activity' bucket for all prefixes that have not been touched, in hardware, for over 85 minutes. Grouping the prefix counts by last-time-used gives us further flexibility in tuning how aggressively we want to move entries to the SFE.

After weeks of data gathering from production SDN ToRs upgraded with the route cache feature, the results were inline with our expectations. Figure 11 shows the hardware table utilization across the route-entry last-hit time bins for the entire SDN ToR fleet. We see that 10% of all the prefixes in the hardware were utilized or 'hit' in the last 100 seconds.

Bluebird does implement an inherent artificial hardware provisioning constraint, to mostly protect against complete hardware table exhaustion on the SDN ToR. This protection was put in place to ensure that the 192K entries are never consumed. However, what we learned was that the hardware table utilization is in fact only 50% utilized and no more than 20-25% of the prefixes are active at any given time (Figure 11). This means that that a majority of the prefixes (75-80%) that are in hardware can in fact be moved to the SFE.

Armed with this data, we can now remove the Bluebird provisioning constraints and allow for provisioning of more than 192K entries knowing that 75% of the prefixes will most likely reside in the SFE.

Based on Figure 11, we can conclude that entries categorized under 'no activity' are good candidates to migrate to

the SFE leaving newly-vacated hardware entries available to other customers. Since only 20-25% of the entries are used at any given time, route caching allows us to increase our scale by 4-5x.

6 Operationalization and Experiences

Bluebird has now been deployed at scale in multiple data centers for various bare-metal workloads. The service has brought together high-throughput and low-latency bare-metal offerings to existing cloud customers without compromising scale or reliability. Bluebird has accomplished all the goals that we set out in §3. In order to make Bluebird successful, we adopted well-known operational models including continuous integration for both service delivery and feature development, ensuring redundancy in all aspects, planned failure for maintenance purposes, and incorporating monitoring and alerting.

During the feature development phase we used a P4 emulator [2] to simulate the entire pipeline in software which gave us a glimpse into the complete lifecycle of a packet. Having this flexibility removed the need for hardware at every stage of the development cycle. The software tools helped implement all the SDN ToR features, simulated the hardware, and allowed for rapid prototyping without any of the usual and costly hardware resource dependencies. Furthermore, P4 provided the flexibility of software with hardware-level performance. For example, routing look-up decisions would generally occur at a certain, fixed point in an ASIC's pipeline. In the case of P4, we had the flexibility of doing a routing look-up after the parser stage or in the MAU, giving us the luxury of normalizing the contents of an incoming packet and acting on any portion of the inner or outer IP header (see Figure 4). It was this flexibility in P4 that also allowed us to limit the UDP source-port values described earlier.

For workloads where data-plane redundancy was required, we paired two SDN ToRs using Multi-chassis Link Aggregation Group (MLAG) (Figure 12). While MLAG seemed like a natural choice for first-hop redundancy at layer-2, we did not want BBS to be concerned with the details of MLAG itself or make MLAG a design requirement moving forward. Hence, we created a common anycast loopback IP to represent both members within a redundant pair of SDN ToRs. BBS configures each member like any other SDN-ToR using a shared anycast loopback IP address representing the SDN ToR's physical address. If traffic were to arrive on a member of the SDN ToR pair with failed links to the bare-metal server, MLAG would locally switch the packets to the neighboring SDN ToR with active links to the bare-metal host.

While the P4 emulator tools helped with the development of the software features running on the SDN ToR, a different software emulator was used to help operationalize the SDN ToR in the network. A Docker container emulating a complete SDN ToR was used to speed up the management plane bring-up between Bluebird and the SDN ToR. Since the
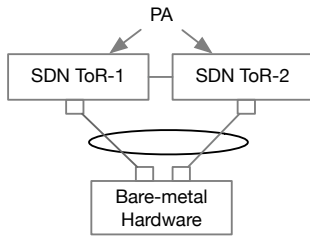
Figure 12: Bare-metal redundancy.

container version of the SDN ToR had all the properties of the hardware based SDN ToR (minus the hardware), the Bluebird management plane was developed and tested against this container. The Docker container was also useful when testing new customer scenarios before taking them to production.

For monitoring and alerting, we built our system to gather numerous metrics per ToR, BM server, and BBS. These metrics are collected in a centralized monitoring system, combined, and further transformed to create alerts. The actions based on the collected metrics differ based on the conditions or events configured. The metrics and alerts are also frequently added and updated as more experience is gained from production. Additionally, they are aggregated in customizable dashboards used to drive decision-making.

6.1 Lessons Learned

These are some of the lessons learned since the release of Bluebird:

- **Data-plane packet mirroring for debugging:** Having the ability to inspect data plane traffic during troubleshooting proved helpful. All data plane packets can be mirrored to the ToR CPU to inspect traffic entering or leaving the SDN ToR. This provides an additional layer of visibility to all the interface and protocol-level counters that are available on the switch. We did not expect this feature would be used as often as it has for debugging issues in production.
- **ASIC with re-configurable programming pipeline:** The Tofino ASIC, with its re-configurable pipeline, allowed us to develop features like route cache even after Bluebird was deployed. The VXLAN source port offset feature, described in §4.3, was possible because of the malleability of the pipeline. Making the decision to use a P4 ASIC proved to be useful as it allowed us to develop all the features that would otherwise have not been possible.
- **ASIC emulators can be used to speed up software development on ToR:** We used a P4 emulator during the ToR packet pipeline development process. Because of this, the development team did not have to wait for the hardware to be available. Moreover, the end-to-end packet flow could be tested by generating actual data-plane packets. This enabled us to test smaller parts of the P4 code without having an end-to-end pipeline ready.

- **Virtualized ToR image for control-plane testing:** Since the availability of the hardware ToR switch for lab testing and development is limited, having a VM image of the ToR was extremely useful to test the route programming service along with the control-plane interaction.
- **Need for 64-bit OS:** With a 32-bit OS, only 4GB of virtual memory could be addressed giving us 600k route cache entries. Hence, moving to a 64-bit OS and increasing the amount of RAM on the SDN-ToR was required to support 1.5M route-cache entries.
- **Limited control-plane vocabulary:** We limited the scope of commands that BBS was allowed to execute on the SDN ToR. The commands issued by BBS are strictly related to adding/deleting customer VRFs and mappings. This limits the damage that can be caused by bad actors and avoids interference with the rest of the automation framework. All other provisioning, maintenance, and monitoring functions are performed by the larger automation framework.
- **Software coordination at scale:** BBS runs on a server-class machine that is far more capable than the comparatively underpowered ToR's hardware. This difference is also reflected in the number of concurrent connections possible, which, if left unchecked, can impact programming time due to BBS exceeding the ToR connection limit. Consequently, requests from BBS are sent to a queue and batched to keep the number of connections within the limits of the switch.
- **Customer traffic should be agnostic of ToR availability:** MLAG helped us abstract out whether a ToR was in maintenance or not. This made the BBS less disruptive to deploy and maintain for redundant workloads.
- **Reconciliation is necessary:** Restoring an outdated ToR configuration can lead to failures and programming conflicts. Reconciliation is necessary to ensure that errors introduced by outdated configurations are repaired. A reconciliation process running after a configuration is restored guarantees eventual consistency and transient conflicts. For instance, CA-to-PA mappings received from upstream are compared with the ToR's current configuration, and stale mappings are removed in the process. Reconciliation is also performed against BBS' internal cache and state. Restoring a state after a service fail-over is also a source of incoherent configurations. Missed notifications during downtime are, in fact, a common occurrence in a live service.
- **Artificial limits can cause overheads:** During analysis of scale requirements from production data on the size of VNETs, we concluded that we had to limit the per-customer mappings until the route cache feature was enabled. As a result, an upper limit was put to stop one customer from monopolizing an entire ToR. This artificial limit soon became an operational overhead since we had to increase it per customer on an on-demand basis and in many cases the new limit was barely above the imposed value.
- **State to reduce reconciliation time:** BBS initially was

developed as a fully stateless service. After each restart, the switch configuration would simply be reconstructed through data received from upstream and downstream components. However, this significantly increased the re-hydration and reconciliation time. Eventually, we moved to a stateful model to reduce the time consuming interactions with the other components. In the new model, BBS maintains a versioned representation of the switch configuration and communication is established only when strictly necessary.

- **For bug fixes, prefer a new image over a patch:** During the initial deployment stages, for quick bug fixes, we deployed bug-fix patches on top of the ToR OS image. But as the number of ToRs grew, it became cumbersome to deploy patches throughout the fleet of devices and keep a track of them. So, we decided to have bug fixes in new images only. Now we upgrade the devices more often but in a manner that is easier to track. This decision has helped us improve the quality of the code overall.
- **Unmodified Linux kernel:** The ToR OS uses an unmodified Linux kernel. Because of this, we were able to use open source tools like tcpdump, iperf, etc. for debugging without any issues. Also, we are able to run Docker containers on top of the ToR OS for SSH user certificate rotation.

7 Related Work

A practical implementation of Bluebird relies on the ability to enable custom and dynamic SDN policies in a ToR, enabled by recent work in programmable switches [4, 5]. As discussed in §2, many other forms of hardware can be used to implement the SDN stack including smartNICs [58–60] and servers running any one of a variety of software network processing systems such as [17, 25, 41, 62]. While there is a vast array of prior work in this space, the state-of-the-art software solutions are not able to meet the throughput of a programmable switch and require far more power. Work in network function virtualization [20, 43, 49, 54, 66, 68, 70] shows that these software-based approaches can be feasible at scale, though they do not meet our stringent requirements. Similarly, smartNICs have been used to offload various custom network operations [15, 44]. However, the bare-metal model precludes these options as they reside at the host. One may also adopt a hybrid approach, leveraging commodity switches and software [3, 18], but again the power consumption of a server is higher than a ToR switch.

Programmable switches have been used for a wide variety of other applications such as caching [32, 45, 47], telemetry [24, 57], consensus [11, 31], machine learning [63], and various network functions [37, 39, 52]. Despite the well-known and strict resource constraints in programmable switches [65], these systems demonstrate that non-trivial computations can be done in the network at line rate on these devices.

Bluebird leverages this speed while overcoming the state limitations of Tofino switches by using the switch CPU and memory for cached flows. As the scale of the necessary state continues to grow, upgrading to the Tofino-2 [28] or adopting a switch memory extension may be helpful [40]. With increased traffic engineering and the rise of SDN, the limits of in-switch memory have become a noticeable issue prompting investigations into the practicality of caching for traditional routes [38, 46] and flow policies [9, 36]. We do not have to implement the complex dependency logic of [36] since the CA-PA mappings are non-overlapping. Similarly, [38, 46] and [9] capitalize on relationships between entries in a FIB or openflow table [51] which are not present in Bluebird's in-switch data. Other SDN rule distribution techniques [34, 35, 55] do not apply to the Bluebird design as there is only one switch on route to implement the necessarily policies.

Since the early and largely academic SDN designs [7, 19, 22, 23, 42, 51], hyperscale cloud networks have adopted SDN to virtualize and isolate tenant networks [10, 12, 14, 21, 30, 56, 61], implementing at various layers of the stack. Regardless of multi-tenancy, SDN is used to operate large single-tenant networks with high-level intent and to implement arbitrary traffic engineering [8, 26, 29, 64, 67, 69, 71]. Bluebird is a new addition to our SDN deployment accommodating the unique requirements of baremetal customers: the servers must be connected to virtual networks, but we cannot enforce any SDN policies at the host in an approach such as [50]. This encourages new in-ToR support so that the additional costs of running the necessary network functions on a server similarly to [50] can be avoided.

8 Conclusions and Future Work

We have presented our experiences designing, implementing, and deploying Bluebird, a high-performance network virtualization system for bare-metal cloud services on Azure. Bluebird has been running in Azure data centers for more than two years and has served demanding workloads like those for Netapp, Cray, and SAP.

The abstraction layer in Bluebird's control plane allows us to handle different switches with minimal change. By using high-performance programmable ASICs, we rearranged ToR's resources to increase route capacity. On top of that, we have implemented a cache system that extends the capacity even further while incurring a negligible performance penalty. The support for additional routes has allowed us to improve performance by removing software gateways. Lastly, our design decouples the SDN stack from the bare-metal services and facilitates the introduction of new and diverse workloads.

In the future, as we learn more about customer traffic, we will explore ways to improve the cache system, such as by considering a different eviction algorithm.

References

[1] https://github.com/aristanetworks/
p4-vxlanencapdecap/blob/main/switch-vxlan.p4.

[2] P4 behavioral model, 2021. https://github.com/p4lang/
behavioral-model.

[3] Mina Tahmasbi Arashloo, Pavel Shirshov, Rohan Gandhi, Guo-han Lu, Lihua Yuan, and Jennifer Rexford. A scalable VPN gateway for multi-tenant cloud services. *SIGCOMM Comput. Commun. Rev.*, 48(1):49–55, April 2018.

[4] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKe-own, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[5] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *Proceed-ings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 99–110. Association for Computing Machinery, 2013.

[6] Broadcom. Trident SmartToR, 2021. https://www.
broadcom.com/products/ethernet-connectivity/
switching/strataxgs/smarttor.

[7] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking con-trol of the enterprise. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '07, page 1–12. Association for Computing Machinery, 2007.

[8] Sean Choi, Boris Burkov, Alex Eckert, Tian Fang, Saman Kazemkhani, Rob Sherwood, Ying Zhang, and Hongyi Zeng. FBOSS: Building switch software at scale. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 342–356. Associ-ation for Computing Machinery, 2018.

[9] Andrew R. Curtis, Jeffrey C. Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. DevoFlow: Scaling flow management for high-performance networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIG-COMM '11, page 254–265. Association for Computing Ma-chinery, 2011.

[10] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Are-fin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, En-rique Cauich Zermeno, Erik Rubow, James Alexander Docauer, Jesse Alpert, Jing Ai, Jon Olson, Kevin DeCabooter, Marc de Kruijf, Nan Hua, Nathan Lewis, Nikhil Kasinadhuni, Ric-cardo Crepaldi, Srinivas Krishnan, Subbaiah Venkata, Yossi Richter, Uday Naik, and Amin Vahdat. Andromeda: Perfor-mance, isolation, and velocity at scale in cloud network virtu-alization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 373–387, Ren-ton, WA, April 2018. USENIX Association.

[11] Huynh Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Noa Zilberman, Hakim Weatherspoon, Marco Canini, Fernando Pedone, and Robert Soulé. P4xos: Consensus as a network service. *IEEE/ACM Trans. Netw.*, 28(4):1726–1738, August 2020.

[12] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Ma-glev: A fast and reliable software network load balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 523–535, Santa Clara, CA, March 2016. USENIX Association.

[13] Hewlett Packard Enterprise. Cray clusterstor e1000 storage systems, 2021. https://buy.hpe.com/us/
en/enterprise-solutions/storage-solutions/
cray-clusterstor-storage-systems/
cray-clusterstor-e1000-storage-systems/
cray-clusterstor-e1000-storage-systems/p/
1012842049.

[14] Daniel Firestone. VFP: A virtual switch platform for host SDN in the public cloud. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 315–328, Boston, MA, March 2017. USENIX Association.

[15] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Har-ish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure accelerated networking: SmartNICs in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, Renton, WA, April 2018. USENIX Association.

[16] Linux Foundation. Open vSwitch, 2016. https:
//www.nvidia.com/en-us/networking/ethernet/
connectx-5/.

[17] Linux Foundation. Data plane development kit (DPDK), 2021. http://www.dpdk.org.

[18] Rohan Gandhi, Hongqiang Harry Liu, Y. Charlie Hu, Guo-han Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. Duet: Cloud scale load balancing with hardware and software. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIG-COMM '14, page 27–38. Association for Computing Machin-ery, 2014.

[19] Yashar Ganjali and Amin Tootoonchian. HyperFlow: A dis-tributed control plane for OpenFlow. In *2010 Internet Net-work Management Workshop/Workshop on Research on Enter-prise Networking (INM/WREN 10)*, San Jose, CA, April 2010. USENIX Association.

[20] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. OpenNF: Enabling innovation in network function control. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 163–174. Association for Computing Machinery, 2014.

[21] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: A scalable and flexible data center network. *Commun. ACM*, 54(3):95–104, March 2011.

[22] Albert Greenberg, Gisli Hjalmtysson, David A. Maltz, Andy Myers, Jennifer Rexford, Geoffrey Xie, Hong Yan, Jibin Zhan, and Hui Zhang. A clean slate 4D approach to network control and management. 35(5):41–54, October 2005.

[23] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. NOX: Towards an operating system for networks. *SIGCOMM Comput. Commun. Rev.*, 38(3):105–110, July 2008.

[24] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 357–371. Association for Computing Machinery, 2018.

[25] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. MegaPipe: A new programming interface for scalable network I/O. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, page 135–148, USA, 2012. USENIX Association.

[26] Chi-Yao Hong, Subhasree Mandal, Mohammad Al-Fares, Min Zhu, Richard Alimi, Kondapa Naidu B., Chandan Bhagat, Sourabh Jain, Jay Kaimal, Shiyu Liang, Kirill Mendelev, Steve Padgett, Faro Rabe, Saikat Ray, Malveeka Tewari, Matt Tierney, Monika Zahn, Jonathan Zolla, Joon Ong, and Amin Vahdat. B4 and after: Managing hierarchy, partitioning, and asymmetry for availability and scale in Google's software-defined WAN. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 74–87. Association for Computing Machinery, 2018.

[27] Intel. Intel Tofino, 2021. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html.

[28] Intel. Intel Tofino 2, 2021. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html.

[29] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined WAN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 3–14. Association for Computing Machinery, 2013.

[30] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Albert Greenberg, and Changhoon Kim. EyeQ: Practical network performance isolation at the edge. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 297–311, Lombard, IL, April 2013. USENIX Association.

[31] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. NetChain: Scale-free sub-RTT coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 35–49, Renton, WA, April 2018. USENIX Association.

[32] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 121–136. Association for Computing Machinery, 2017.

[33] Gopal Kakivaya, Lu Xun, Richard Hasha, Shegufta Bakht Ahsan, Todd Pfleiger, Rishi Sinha, Anurag Gupta, Mihail Tarta, Mark Fussell, Vipul Modi, Mansoor Mohsin, Ray Kong, Anmol Ahuja, Oana Platon, Alex Wun, Matthew Snider, Chacko Daniel, Dan Mastrian, Yang Li, Aprameya Rao, Vaishnav Kidambi, Randy Wang, Abhishek Ram, Sumukh Shivaprakash, Rajeet Nair, Alan Warwick, Bharat S. Narasimman, Meng Lin, Jeffrey Chen, Abhay Balkrishna Mhatre, Preetha Subbarayalu, Mert Coskun, and Indranil Gupta. Service fabric: A distributed platform for building microservices in the cloud. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18. Association for Computing Machinery, 2018.

[34] Nanxi Kang, Zhenming Liu, Jennifer Rexford, and David Walker. Optimizing the one big switch abstraction in software-defined networks. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, page 13–24. Association for Computing Machinery, 2013.

[35] Y. Kanizo, D. Hay, and I. Keslassy. Palette: Distributing tables in software-defined networks. In *2013 Proceedings IEEE INFOCOM*, pages 545–549, 2013.

[36] Naga Katta, Omid Alipourfard, Jennifer Rexford, and David Walker. CacheFlow: Dependency-aware rule-caching for software-defined networks. In *Proceedings of the Symposium on SDN Research*, SOSR '16. Association for Computing Machinery, 2016.

[37] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. HULA: Scalable load balancing using programmable data planes. In *Proceedings of the Symposium on SDN Research*, SOSR '16. Association for Computing Machinery, 2016.

[38] Changhoon Kim, Matthew Caesar, Alexandre Gerber, and Jennifer Rexford. Revisiting route caching: The world should be flat. In *Proceedings of the 10th International Conference on Passive and Active Network Measurement*, PAM '09, page 3–12, Berlin, Heidelberg, 2009. Springer-Verlag.

[39] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. TEA: Enabling state-intensive network functions on programmable switches. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 90–106. Association for Computing Machinery, 2020.

[40] Daehyeok Kim, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, and Srinivasan Seshan. Generic external memory for switch data planes. In *Proceedings of the 17th ACM Workshop on Hot*

Topics in Networks, HotNets '18, page 1–7. Association for Computing Machinery, 2018.

[41] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, August 2000.

[42] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: A distributed control platform for large-scale production networks. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, page 351–364, USA, 2010. USENIX Association.

[43] Chang Lan, Justine Sherry, Raluca Ada Popa, Sylvia Ratnasamy, and Zhi Liu. Embark: Securely outsourcing middleboxes to the cloud. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 255–273, Santa Clara, CA, March 2016. USENIX Association.

[44] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. ClickNP: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 1–14. Association for Computing Machinery, 2016.

[45] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. Incbricks: Toward in-network computation with an in-network cache. *SIGARCH Comput. Archit. News*, 45(1):795–809, April 2017.

[46] Yaoqing Liu, Syed Obaid Amin, and Lan Wang. Efficient FIB caching using minimal non-overlapping prefixes. *SIGCOMM Comput. Commun. Rev.*, 43(1):14–21, January 2013.

[47] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. DistCache: Provable load balancing for large-scale storage systems with distributed caching. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 143–157, Boston, MA, February 2019. USENIX Association.

[48] Mallik Mahalingam, Dinesh Dutt, Kenneth Duda, Puneet Agarwal, Larry Kreeger, T. Sridhar, Mike Bursell, and Chris Wright. Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks. RFC 7348, August 2014.

[49] Joao Martins, Mohamed Ahmed, Costin Raiciu, and Felipe Huici. Enabling fast, dynamic network processing with ClickOS. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, page 67–72. Association for Computing Machinery, 2013.

[50] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 399–413. Association for Computing Machinery, 2019.

[51] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[52] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. SilkRoad: Making stateful layer-4 load balancing fast and cheap using switching ASICs. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 15–28. Association for Computing Machinery, 2017.

[53] Microsoft. Azure NetApp files, 2021. https://azure.microsoft.com/en-us/services/netapp/#overview.

[54] R. Mijumbi, J. Serrat, J. Gorricho, N. Bouten, F. De Turck, and R. Boutaba. Network function virtualization: State-of-the-art and research challenges. *IEEE Communications Surveys Tutorials*, 18(1):236–262, 2016.

[55] Masoud Moshref, Minlan Yu, Abhishek Sharma, and Ramesh Govindan. Scalable rule management for data centers. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 157–170, Lombard, IL, April 2013. USENIX Association.

[56] Jayaram Mudigonda, Praveen Yalagandula, Jeff Mogul, Bryan Stiekes, and Yanick Pouffary. NetLord: A scalable multi-tenant network architecture for virtualized datacenters. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, page 62–73. Association for Computing Machinery, 2011.

[57] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 85–98. Association for Computing Machinery, 2017.

[58] NVIDIA. Connectx-5, 2021. https://www.nvidia.com/en-us/networking/ethernet/connectx-5/.

[59] NVIDIA. Data processing units, 2021. https://www.nvidia.com/en-us/networking/products/data-processing-unit/.

[60] NVIDIA. Innova-2 flex, 2021. https://www.nvidia.com/en-us/networking/ethernet/innova-2-flex/.

[61] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. Ananta: Cloud scale load balancing. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 207–218. Association for Computing Machinery, 2013.

[62] Luigi Rizzo and Matteo Landi. Netmap: Memory mapped access to network devices. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, page 422–423. Association for Computing Machinery, 2011.

[63] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan R. K. Ports, and Peter Richtárik. Scaling

distributed machine learning with in-network aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, April 2021.

[64] Brandon Schlinker, Hyojeong Kim, Timothy Cui, Ethan Katz-Bassett, Harsha V. Madhyastha, Italo Cunha, James Quinn, Saif Hasan, Petr Lapukhov, and Hongyi Zeng. Engineering egress with edge fabric: Steering oceans of content to the world. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 418–431. Association for Computing Machinery, 2017.

[65] Naveen Kr. Sharma, Antoine Kaufmann, Thomas Anderson, Arvind Krishnamurthy, Jacob Nelson, and Simon Peter. Evaluating the power of flexible packet processing for network resource allocation. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 67–82, Boston, MA, March 2017. USENIX Association.

[66] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. *SIGCOMM Comput. Commun. Rev.*, 42(4):13–24, August 2012.

[67] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Hong Liu, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter rising: A decade of Clos topologies and centralized control in Google's datacenter network. *Commun. ACM*, 59(9):88–97, August 2016.

[68] Arjun Singhvi, Junaid Khalid, Aditya Akella, and Sujata Banerjee. SNF: Serverless network functions. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 296–310. Association for Computing Machinery, 2020.

[69] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky H.Y. Wong, and Hongyi Zeng. Robotron: Top-down network management at Facebook scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 426–439. Association for Computing Machinery, 2016.

[70] Richard Wang, Dana Butnariu, and Jennifer Rexford. OpenFlow-based server load balancing gone wild. In *Proceedings of the 11th USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, Hot-ICE'11, page 12, USA, 2011. USENIX Association.

[71] Kok-Kiong Yap, Murtaza Motiwala, Jeremy Rahe, Steve Padgett, Matthew Holliman, Gary Baldus, Marcus Hines, Taeeun Kim, Ashok Narayanan, Ankur Jain, Victor Lin, Colin Rice, Brian Rogan, Arjun Singh, Bert Tanaka, Manish Verma, Puneet Sood, Mukarram Tariq, Matt Tierney, Dzevad Trumic, Vytautas Valancius, Calvin Ying, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Taking the edge off with Espresso: Scale, reliability and programmability for global internet peering. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 432–445. Association for Computing Machinery, 2017.

Cetus: Releasing P4 Programmers from the Chore of Trial and Error Compiling

Yifan Li‡,†, Jiaqi Gao†, Ennan Zhai†, Mengqi Liu†, Kun Liu†, Hongqiang Harry Liu†
‡Tsinghua University   †Alibaba Group

Abstract

Programmable switches are widely deployed in Alibaba's edge networks. To enable the processing of packets at line rate, our programmers use P4 language to offload network functions onto these switches. As we were developing increasingly more complex offloaded network functions, we realized that our development needs to follow a certain set of constraints in order to fit the P4 programs into available hardware resources. Not adhering to these constraints results in *fitting issues*, making the program uncompilable. Therefore, we decide to build a system (called Cetus) that automatically converts an uncompilable P4 program into a functionally identical but compilable P4 program. In this paper, we share our experience in the building and using of Cetus at Alibaba. Our design insights for this system come from our investigation of the past fitting issues of our production P4 programs. We found that the long dependency chains between actions in our production P4 programs are creating difficulties for the programs to comply with the hardware resources of programmable switching ASICs, resulting in the majority of our fitting issues. Guided by this finding, we designed the core approach of Cetus to efficiently synthesize a compilable program by shortening the lengthy dependency chains. We have been using Cetus in our production P4 program development for one year, and it has effectively decreased our P4 development workload by two orders of magnitude (from O(day) to O(min)). In this paper we share several real cases addressed by Cetus, along with its performance evaluation.

1 Introduction

Programmable switches allow network programmers to use P4 language to offload network functions to data planes, enabling these functions to process packets at line rate. As one of the largest global service providers, Alibaba has widely deployed programmable switches in its edge networks [20, 27]. By Jan 2021, we have built O(100) PoP (point of presence) nodes and O(1000) edge sites in total, and the majority of them have employed programmable switches to implement a group of network functions, including firewall, DDoS defense, and load balancer. Figure 1 shows an example of the architecture of network functions within a single programmable switch in our edge networks. In this architecture, our programmers offload multiple network functions to a single programmable switch, enabling these network functions to process packets at Tbps speeds and saving CPU resources on the end-servers in edge networks.

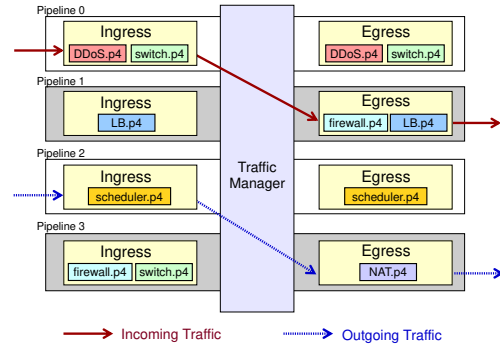While our business significantly benefits from the deploy-

Figure 1: A gateway P4 program example deployed in Alibaba's edge network. In our edge network scenario, our programmers put various network functions in a single switch.

ment of programmable switches, nevertheless, we still encounter a tough problem. Our P4 program development—*e.g.*, implementation of new network functions and update of the existing network functions via P4—needs to take into account the various constraints of programmable switching ASICs; neglecting these constraints often results in programs that cannot fit on the hardware and hence cannot compile. We call this problem as *fitting issue*.

Fitting a P4 program is hard to our programmers, because (1) programmable switching ASICs have various hardware resources, each with unique size and constraints, and (2) resources are sometimes correlated, reducing the resource A usage of a program coming at the cost of increasing the usage of resource B. Our programmers, therefore, usually fall into time consuming trial and error program "reshaping" cycles, significantly delaying their development time. On the other hand, it is impractical to require our programmers to learn all hardware constraints.

Alibaba therefore decided to build a system (called Cetus) that automatically converts an uncompilable P4 program P into a functionally identical but compilable P4 program P'.

State of the art. Existing work falls into two categories. On the one hand are systems that compile a high-level abstraction to generate optimized P4 programs [10, 13, 14, 25, 30]. Although they offer good resource optimizations, we found these solutions may not be effective in our specific scenario. For example, P4All [13, 14] optimizes the resource usage among network functions by explicitly leveraging reusable data structures (*e.g.*, bloom filters and key-value stores); however, the network functions within our production P4 programs do not share these data structures, invalidating this optimization in our case. In addition, our programmers are reluctant to use

an extension of P4 such as explicitly specifying some data structure to optimize via `objective` in P4All. Another state-of-the-art system, Lyra [10], merges the tables that have no dependencies with each other in order to optimize the resource usage; however, we found that merging tables while keeping the original dependencies is not enough to enable our production P4 programs to fit into the programmable ASICs. On the other hand, existing efforts like Chipmunk [11, 12] and Domino [24] improve P4 compiler to synthesize optimized switch binary code, which is different from our goal of generating optimized P4 programs.

Our approach: Cetus. This paper shares our experience in the building and using of Cetus at Alibaba. We first investigated our production P4 programs and their past fitting issues, in order to derive insight for our solution design. We found that the **long** dependency chains between actions in our production P4 programs were creating difficulties for the programs to comply with the hardware resources of programmable switching ASICs, resulting in the majority of fitting issues.

Guided by the above finding, we designed Cetus. For a given P4 program P, Cetus automatically merges tables to fit into fewer stages by removing dependencies between tables, thus shortening the long dependency chains (§5). Because such a method may generate many table merging options (called candidates), we propose an approach, called constraint-based filter & optimizer (§6), to drop the candidates that do not satisfy hardware resources (including memory size, PHV, and crossbar) or constraints, and then select the best one as P'. Designing such a filter & optimizer approach is non-trivial due to two challenges: (1) the large formula encoding each candidate may result in state explosion, and (2) large solution searching space in each candidate will cause long solving time. We propose PHV sharing encoding (§6.1) and two-step solving (§6.2) to address the above two challenges, respectively. With P' in hand, Cetus automatically generates a set of control plane APIs for P' to enable P' to be deployed seamlessly (§7).

Finally, we share several representative real cases addressed by Cetus (§8), along with its performance evaluation (§9). We have been using Cetus in production for one year, and it has effectively decreased our P4 development workload by two orders of magnitude (from $O(\text{day})$ to $O(\text{min})$).

2 Preliminary: Programmable Data Plane

We use Υ to denote the name of programmable switching ASICs of Vendor A.[1] Our programmers compile P4 programs via Υ compiler. Υ chip is a physical implementation of *Protocol Independent Switch Architecture* (or PISA). Υ chip's ingress and egress consist of 12 stages, respectively. All of these stages are identical, in terms of compute units, memory types, and memory capacities.

---

[1]We omit the vendor name and ASIC name for the confidentiality.

2.1 Hardware & Constraints of Υ Chip

Hardware resource. Υ chip contains various hardware resources, and each of them has unique size and characteristic. We are mainly focused on the following hardware resources:

- **Pipeline stages.** The packet processing pipeline consists of a fixed number of individual stages. A P4 program does not compile if it takes more than 12 stages in an ingress or egress pipeline in Υ chip.

- **Packet header vector (PHV).** The PHV is a "bus" that carries information (from packet fields and per-packet metadata) between stages. PHV cannot carry more data than its total width. See §6.1 for more PHV details.

- **Memory.** Memory resources mainly contain SRAM and TCAM. SRAM and TCAM are around tens of Megabytes in capacity. The memory resources are equally split and attached to each stage so that each stage can only access its local memory resources.

- **Crossbar.** In each stage, the crossbar extracts fields from the PHV and sends them to the match and action units for computation. Crossbar has a size limit, so the total number of bytes assigned to a stage's crossbar should not exceed this limit.

Hardware constraints. The hardware constraints, in this paper, refer to both the hardware resource characteristics (*e.g.*, in Υ chip, memories are stage local, and memory can be accessed no more than once per packet), and the mappings between the P4 program elements and hardware resources (*e.g.*, a P4 table's keys should be stored in SRAM or TCAM memory, and a packet header field should be mapped into one or multiple cells in the PHV). Understanding these hardware constraints is crucial to programming on the Υ chip.

To successfully compile a P4 program via Υ compiler, this program must not exceed the size of each hardware resource and comply with all constraints of Υ chip.

Fitting a P4 program in our practice. Our production P4 programs typically pack as many functions and modules as possible, which may overuse hardware resources or violate the hardware constraints, resulting in the fitting issues. When this happens, our programmers have to 'reshape' the programs to fit into the programmable ASIC. Such a reshaping process is program specific. Our programmers often spend a significant amount of time reshaping our P4 programs in order to comply with the hardware resources and constraints.

2.2 Dependencies between Tables

A P4 program is a collection of match-action tables chained together by branching conditions. In each table, at most one action can be applied according to the match result. For a given group of actions, if there is no read-write or write-write dependency among these actions, they could be placed within the same stage. On the contrary, for example, if action i_1 uses (reads or writes) a value generated by action i_2, then i_1 must
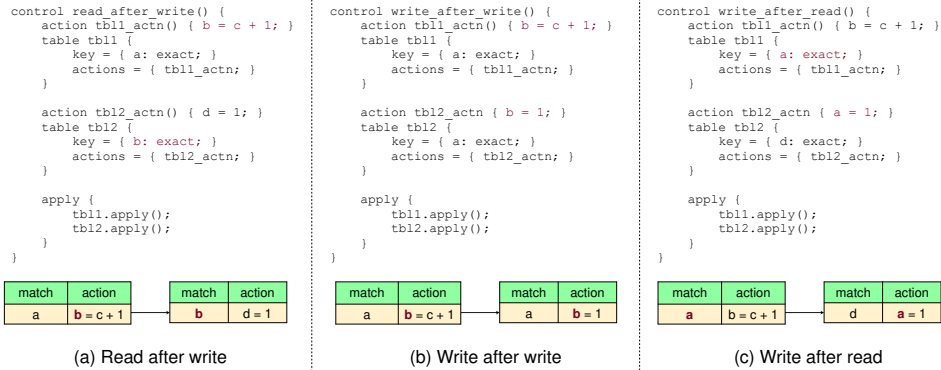
```
control read_after_write() {
    action tbl1_actn() { b = c + 1; }
    table tbl1 {
        key = { a: exact; }
        actions = { tbl1_actn; }
    }

    action tbl2_actn() { d = 1; }
    table tbl2 {
        key = { b: exact; }
        actions = { tbl2_actn; }
    }

    apply {
        tbl1.apply();
        tbl2.apply();
    }
}
```

```
control write_after_write() {
    action tbl1_actn() { b = c + 1; }
    table tbl1 {
        key = { a: exact; }
        actions = { tbl1_actn; }
    }

    action tbl2_actn { b = 1; }
    table tbl2 {
        key = { a: exact; }
        actions = { tbl2_actn; }
    }

    apply {
        tbl1.apply();
        tbl2.apply();
    }
}
```

```
control write_after_read() {
    action tbl1_actn() { b = c + 1; }
    table tbl1 {
        key = { a: exact; }
        actions = { tbl1_actn; }
    }

    action tbl2_actn { a = 1; }
    table tbl2 {
        key = { d: exact; }
        actions = { tbl2_actn; }
    }

    apply {
        tbl1.apply();
        tbl2.apply();
    }
}
```

(a) Read after write (b) Write after write (c) Write after read

Figure 2: Three types of dependencies between actions in our production P4 programs.

(a) The match-action DAG for the original P4 program

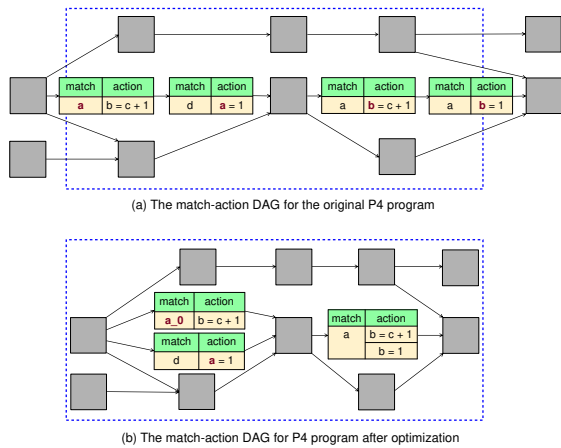(b) The match-action DAG for P4 program after optimization

Figure 3: Examples for match-action DAGs. Rectangles represent tables. The blue dashed frame represents the architecture of Υ chip. The blue dashed frame's length and width represent the usages of stage and memory, respectively, in Υ chip. (a) shows a match-action DAG representing a given P4 program P. P does not fit in Υ chip. (b) is a match-action DAG representing P' that tweaked from P, which is compilable.

be placed in a stage after the stage of i_2 in the PISA architecture. In our production P4 programs, we are mainly focused on three types of dependencies: read after write, write after write and write after read[2]. Figure 2 shows their examples. The tables, in Figure 2(a), (b), and (c), are not allowed to be *directly* placed within the same stage; otherwise, the programs' function logic is changed.

Match-action DAG. By tracking dependencies between actions, we can represent a P4 program in the form of a match-action directed acyclic graph or *match-action DAG*. Figure 3(a) presents such a match-action DAG.

Diameter of a match-action DAG. The total number of stages occupied by a P4 program P cannot be less than the *diameter* of the match-action DAG representing P. The diameter of a match-action DAG G is: the number of tables in the longest dependency chain (*i.e.*, the dependency chain containing the highest number of tables) in G. For example

---

[2] We explain why write after write dependency is necessary in §5.1

Figure 4: Our production P4 programs and their involved network functions as well as their diameters. These three programs have been deployed on almost all the programmable switches in our edge networks.

in Figure 3(a), the diameter is 7, because there are 7 tables in the longest dependency chain of the DAG. The diameter in Figure 3(b) is 5. Thus, we can say that the diameter of a match-action DAG (representing P) must be $\leq$ the number of stages, if P compiles.

3 Key Findings & Solution Intuition

In order to release our programmers from trial and error program-reshaping cycles, we need to understand the root causes resulting in fitting issues during the development of our production programs, thus exploring insights for our solution design. Specifically, we selected three mainstream P4 programs (listed in Figure 4) in our production, which were deployed in almost all the edge switches in Alibaba edge networks. We then selected all fitting issues (of these three programs) that took our programmers more than one hour to resolve, and manually analyzed how they were fixed.

We classified our analysis results into two groups. (1) **Group A:** About 80% of fitting issues were resolved by eliminating or reducing dependencies between tables (*e.g.* by reordering or merging them) that allowed us to take advantage of the parallel nature of the switch architecture. (2) **Group B:** 20% issues were resolved by fixing hardware resources and constraints that programmers were not aware of such as PHV
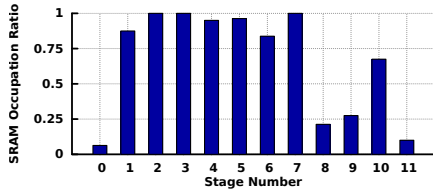
Figure 5: SRAM usage of Edge Gateway program.

allocation and stateful ALUs. We now analyze the principles behind Group A (§3.1) and Group B (§3.2).

3.1 Key Findings from Group A

We investigated why rearranging tables can resolve the fitting issues in this group. We found that all of these efforts (*e.g.*, reordering and merging tables) implicitly shortened the P4 programs' diameters. For example, in one of the cases, our programmer unwittingly merged two tables by changing dependencies between their actions (as shown in Figure 7(a) example), and then found that the program compiled. While this programmer did not know the fundamental reason (*i.e.*, shortening the diameter), he succeeded after multiple reshaping cycles.

Observation 1: Diameter is long in our production. Why shortening the diameter can resolve the fitting issue? We found that the match-action DAG representing each of these three P4 programs had long diameters. Given that Υ chip provides 12 stages of match-action units, a long diameter should be reduced in order to make programs fit on Υ chip. As shown in Figure 3(a), blue dashed frame's length and width represent the usages of stage and memory, respectively, in Υ chip. The program's diameter in Figure 3(a) is too long to comply with the stage resource size.

The long diameter results from the large number of packet processing operations required by our diverse edge services. In particular, each of our P4 programs not only needs to insert various metadata into the different types of packet headers, but also filters or forwards them according to a number of service needs. For example, an input packet is first encapsulated with VXLAN, then forwarded based on some condition, next mirrored for traffic statistics and finally checked by ACL as well as distributed by the ECMP. Figure 4 details these three P4 programs' diameters and their involved network functions. All programs shown in Figure 4 have at least a diameter of 8 in ingress, which means they occupy at least 8 stages in the ingress pipeline. It is therefore highly possible to result in fitting issues in Υ chip when new tables are added.

Observation 2: Many available memory resources. We also found that shortening the diameter by tweaking tables, in principle, increases the usage of memory within individual stages, as shown in Figure 3(b). Why did this memory-for-stage method work in our production? We found that both ends of the match-action DAG (tables with 0 in-degree or out-degree) use much less memory, offering flexibility for table tweaking.

At the beginning of the pipeline, our programs need to perform checking and pre-computations such as packet validation, link aggregation group checking, pre-computing hashes, and setting flag based on header's validity; at the end of the pipeline, our programs finalize the packet processing based on the previous matching results, including marking header fields, dropping packets, and encapsulations. All these operations can be easily done in parallel, while at the same time they do not require a lot of table entries; thus, much available memory remains. Figure 4 shows the percentage of memory that both ends of DAG occupy compared with the entire program. If the memory is distributed evenly across the DAG, both ends of the DAG should occupy around 10% of memory each. Figure 5 shows the SRAM occupation ratio per stage of Edge Gateway program (*i.e.*, the third program in Figure 4). We observed that stage 0 and 11 only used less than 10% of memory. The other two programs also follow the same phenomena.

We also observed much available memory in the middle of the pipeline. Figure 5 shows tables at stage 8 and 9 take only 25% of memory. Similar phenomena also occurred in the rest of the two P4 programs listed in Figure 4. This is because, in a network function chain, we typically have a few tables that are small but critical such as a table inserting a mainstream service-shared DSCP value into the packet header as metadata. Such a table (called T) must have (read-write or write-write) dependency relationships with the tables before and after T.

Summary. We now understand that our programmers unwittingly shortened the diameter of their programs by trial and error table (dependency) tweaking, luckily making their programs compile. Examples in Figure 3(a) and (b) illustrate such an intuition. We therefore derive the following key finding.

> **Finding 1:** Long dependency chains between actions in our production P4 programs make the developed programs hard to fit into the programmable ASIC. We thus need to remove dependencies on the "longest path" of DAG to change the original "long, narrow" DAG to a "short, fat" DAG, as shown in Figure 3, in order to enable our developed programs to compile.

3.2 Key Findings from Group B

Fitting issues in Group B were caused by the violation of chip-specific resource size and hardware constraints. For example, because our programmers ignored the size of an individual stage, the program they wrote required the compiler to assign more DRAM within one stage than allowed (otherwise the dependency constraint is violated), resulting in a fitting issue; the same issue also happened for other resources such as hash units. There is no pattern to follow among these root causes. But we noticed that some of the issues in Group B were caused by same constraint violation. This means that our
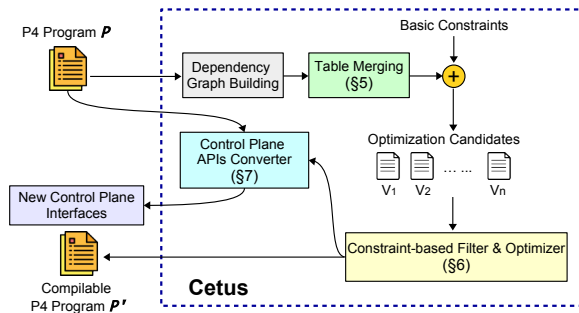
Figure 6: Cetus's workflow overview.

programmers failed to learn or remember the fitting issues that they have ever fixed. We thus derive the following finding.

> **Finding 2:** Although it might be hard for our programmers to learn all chip-specific resource size and constraints, we should avoid the fitting issues—resulting from the unfamiliarity with the resource size and constraints—that we have encountered before.

3.3 Our Solution Idea

Based upon our above two findings, we design the core approach of our solution, which includes the following three steps. First, for a given P4 program P, we automatically merge tables to fit into fewer stages by removing dependencies between actions, in order to shorten the long diameter of DAG representing P (driven by Finding 1). Such an approach would generate many candidate results. Second, we encode hardware resource size and hardware constraints as many as we know in our system's backend DB to ensure that the synthesized program complies with all already-known resource size and constraints (driven by Finding 2). Finally, we check each candidate with the encoded constraints, selecting the most optimal one.

Why the state of the art does not help? Existing systems (*e.g.*, Lyra [10] and P4All [13, 14]) are unable to offer such a level of program optimization. Specifically, Lyra can only merge tables without dependencies. In other words, Lyra cannot merge two tables by removing dependencies between the tables; thus, Lyra is unable to shorten the diameter of the given DAG. P4All optimizes programs by reusing common data structures. In our programs (shown in Figure 4), however, the tables on the diameter do not share any data structure, invalidating P4All's assumption.

4 Cetus's Workflow Overview

We build Cetus, a synthesis system that automatically converts an uncompilable P4 program P into a functionally identical but compilable P4 program P'.

Figure 6 presents Cetus's workflow that consists of the following main phases.

- First, given a P4 program P, Cetus generates a match-action DAG by analyzing read-write and write-write dependencies in P. Then, Cetus introduces a table merging approach (§5) to shorten the diameter of the generated DAG by removing dependencies between tables. There could be many potential table merging cases. We drop the cases that violate basic hardware constraints (*e.g.*, memory size), obtaining a group of candidate programs.

- Second, we propose a constraint-based filter and optimizer (§6) to check each candidate individually with already-known constraints, selecting the most optimal one as P'.

- Finally, Cetus automatically generates a set of control plane APIs for P' to enable P' to be deployed seamlessly (§7).

5 Table Merging by Dependency Removal

Cetus proposes a table merging approach to shorten the diameter by removing dependencies. Intuitively, the purpose of the table merging module is to tweak P to fit into the architecture of Υ chip. This approach includes several primitives to merge tables for different types of dependencies. This section first introduces these primitives (§5.1), and then describes the entire solution (§5.2).

5.1 Dependency Removal Primitives

We design several dependency removal primitives in terms of dependency types, including write-after-write, write-after-read and read-after-write dependencies (shown in Figure 2). Each of the primitives takes two tables as input and returns one or two tables that can be put within one single stage. The purpose of these primitives is to reduce the number of used stages by increasing other resources' overhead such as PHV and memory.

Symbols. We define the following notations: table t has n_m match fields $\{m_{t1}, ..., m_{tn_m}\}$, each field m_{ti} has w_{ti} bits in width and its match type is p_{ti}, which can be *exact*, *ternary*, *etc*. It also has n_a actions $\{a_{t1}, ..., a_{tn_a}\}$. If one table has no default action, we add an empty action as the default. Table t has l_t entries. Let P_t be the action parameters' total bit width, then table t's total memory usage is $l_t(\sum_{i=0}^{n_m} w_{ti} + P_t)$.

Write-after-write (WAW) dependency. WAW dependency happens when one table t_1 contains an action that writes the value written by another table t_2. For example, in Figure 2(b), table `tbl2`'s action *tbl2_actn* writes variable b, which is previously modified by table `tbl1`[3]. Since two actions are not allowed to write to the same data in a PHV word concurrently, one cannot place them in the same stage. It is also impossible to reorder them since the program's correctness is violated.

This primitive removes WAW dependency by merging the two tables into a new table t'. The merged table t' enumerates both tables' all action combinations. The primitive works as follows, and Figure 7(a) shows an example.

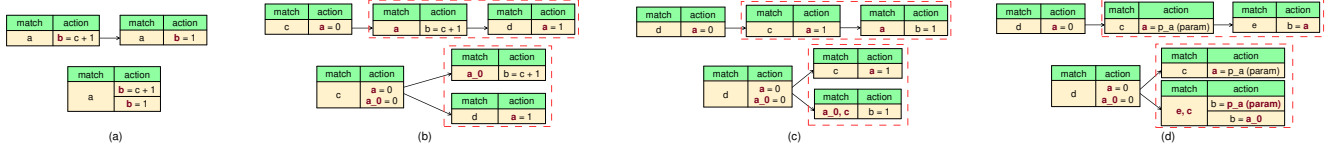[3] We cannot remove `tbl1` because a packet can hit `tbl1` but miss `tbl2`.

Figure 7: Examples for different dependency removal primitives. (a) WAW (b) WAR (c) RAW-match (d) RAW-action

- Merge the match fields of the two tables and generate new match fields $\{m_{t_1 1}, ..., m_{t_1 n_1}, m_{t_2 1}, ..., m_{t_2 n_2}\}$.
- Generate all possible combinations of the two tables' actions $\{(a_{t_1 1}, a_{t_2 1}), (a_{t_1 2}, a_{t_2 1}), ..., (a_{t_1 n}, a_{t_2 n})\}$
- Merge each pair of actions into one by appending the statements in the second action after the first one.
- When a merged action has two statements that write the same value, one from t_1, one from t_2, we keep the latter one.

Memory usage. Since the two tables hit and miss independently, the merged table should include all four possibilities. Thus, unless two tables have identical match fields, table t' uses ternary match field types and is deployed in the TCAM memory. In total, there are $(l_{t_1} + 1)(l_{t_2} + 1) - 1$ entries. The total memory usage of table t' is $l_{t_1} l_{t_2} (\sum_{i=0}^{n_{t_1 m}} w_{t_1 i} + P_{t_1} + \sum_{i=0}^{n_{t_2 m}} w_{t_2 i} + P_{t_2})$.

Write-after-read (WAR) dependency. When one table t_2 writes the variable read by t_1, WAR dependency happens. For example, in Figure 2(c), the table `tbl2`'s action *tbl2_actn* writes variable a, which is table `tbl1`'s match fields. Again, we cannot reorder these two tables; however, PISA architecture allows t_2 to be deployed alongside t_1. When t_1 occupies multiple stages, t_2 can only share t_1's last stage and not earlier. WAR dependency does not necessarily increase the total number of stages of a program directly, but it sets a "barrier" and pushes other tables to later stages. For example, in Figure 2(c), if we have a third table `tbl3` that reads variable a after table `tbl2`, then it has to be deployed after table `tbl1`, even though there is no dependency between `tbl1` and `tbl3`.

For WAR dependency, let x be the shared variable. We have table t_1 reads x and table t_2 writes it. To remove WAR dependency, we create a new copy of the shared variable x' and modify t_1 so that it reads x' instead of x. The primitive works as follows, and Figure 7(b) shows the example.

- Find the table where x is last written. If such a table exists, copy the action that writes x and modifies it to write x'. If no such table exists, such as x is a header, then we assign the value of x' in the parser.
- Modify table t_1's match and action list so that it reads x'.

Memory usage. This primitive does not create a new table and the memory usage is kept the same. It may introduce PHV overhead since it creates a new variable.

Read-after-write (RAW) dependency. Read-after-write dependency happens when one table (t_2) reads the value created by another one (t_1). For example, in Figure 2(a), table `tbl2`'s

match fields read the value written by table `tbl1`'s action. The dependency can also happen when the value is read in the action field. Same as the WAW dependency, two tables with RAW dependency between them have to be placed in different stages and cannot be reordered.

This primitive removes the RAW dependency by summarizing the primitives used in WAW and WAR dependency: we first create a new table t' that summarizes the match fields of both tables and replace t_2, and then we adopt WAR dependency removal primitive to remove the dependency between t_1 and t'.

Let $x = f(\mathbf{v_1})$ be action in t_1 that modifies shared variable x. In Figure 2(a), $\mathbf{v_1}$ is $\{c, 1\}$. Assume the action is executed when table t_1 matches value $\mathbf{v_2}$, then after applying table t_1, x's value is:

$$x = \begin{cases} f(\mathbf{v_1}) & \text{if } (m_{t_1 1}, m_{t_1 2}, ..., m_{t_1 n}) = \mathbf{v_2} \\ x_0 & \text{otherwise} \end{cases} \quad (1)$$

where x_0 is the value of x before applying table t_1. The key of the dependency removal primitive is to encode enough information in a new table t' to compute variable x without using the result in t_1. Equation 1 shows that x depends on three sets of variables $\mathbf{v_1}, \mathbf{v_2}, x_0$. We can learn $\mathbf{v_2}$ from entries in table t_1. x_0 is created before t_1, so we borrow the primitive used in WAR dependency removal and create a new copy of variable x. So our challenge is reduced to understanding $\mathbf{v_1}$.

Theoretically, since variables in $\mathbf{v_1}$ have fixed lengths, we can enumerate all possibilities. However, this would lead to too much memory overhead. As a result, we only remove RAW dependency when we can infer values in $\mathbf{v_1}$ easily, such as when all of them are numbers or assigned to numbers directly. In Figure 2(a), $\mathbf{v_1} = \{c, 1\}$. If we can infer the value of c, then we can merge `tbl1` and `tbl2`, otherwise, we cannot. Cetus removes dependency differently depending on whether table t_2 reads variable x in match or action part. If table t_2 reads x in the match fields, the primitive works as follows, and Figure 7(c) shows the example tables and merged result.

- Create a copy of variable x through the method introduced in the WAR dependency removal primitive, let the copy be x_0.
- Merge the match fields of the two tables, remove x, and generate new match fields $\{m_{t_1 1}, ..., m_{t_1 n_1}, m_{t_2 1}, ..., m_{t_2 n_2}\} - \{x\} + \mathbf{v_1} + \{x_0\}$.
- Remove constants from the match field. For example when $\mathbf{v_1}$ or x_0 is fixed.

	RAW	WAW	WAR
Direct stateful objects	N	N	Y
Normal & not directly involved	Y	Y	Y
Normal & directly involved	N	Y	Y

Table 1: Cetus applies primitives to different cases.

- Generate a new table t' with the new match fields. Copy table t_2's action field to the table t'.

If the table t_2 reads x in the action field, we need to encode both branches in Equation 1 and duplicate actions that read x. The primitive works as follows, Figure 7(d) shows the example tables and merged results.

- Create a copy of variable x through the method introduced in the WAR dependency removal primitive, let the copy be x_0.

- Merge the match fields of the two tables and generate new match field $\{m_{t_1 1}, ..., m_{t_1 n_1}, m_{t_2 1}, ..., m_{t_2 n_2}\} + \nu_1$.

- Remove constants from the match fields.

- For each action $a_{t_2 i}$ that reads x, replace x with new copy x_0. Create a new copy $a'_{t_2 i}$ and add x_0 into its parameter. Action $a'_{t_2 i}$ is triggered when Equation 1's first condition is triggered, Action $a_{t_2 i}$ is triggered when the second condition is triggered.

- New table t' has the new match fields, all actions from table t_2, and newly generated actions $a'_{t_2 i}$.

Memory usage. Memory usage varies depending on how many constants we can infer. Assume we can infer the value of x_0 and ν_1, then the newly generated table t' takes up $l_{t_2}(\sum_{i=0}^{n_{t_1} m} w_{t_1 i} + \sum_{i=0}^{n_{t_2} m} w_{t_2 i} + P_{t_2})$ memory. The newly generated table's match fields stays the same.

Multiple dependencies between two tables. Two tables can have more than one dependency and may not be limited to the same type. For example, they can have WAW and WAR dependency at the same time, or have two RAW dependencies. When dependencies have the same type, we can apply the pre-mentioned primitives directly (WAW) or recursively (RAW, WAR) to remove dependencies. For different dependency types, we choose not to remove them since the result table usually incurs too much memory overhead.

Counters, meters, and registers. In ASIC, stateful objects such as counters have two modes: direct and indirect. Direct counters have one-to-one mapping with table entries, while indirect ones have user-defined sizes. Depending on their mode and whether they are involved in the dependency directly (*i.e.* they write to variables read or written by another table), Cetus chose whether apply different primitives differently, and it is summarized in Table 1.

5.2 Table Merging Approach

Given a P4 program with n dependencies, there could be 2^n different table merging strategies at most. Different strategies produce different resource-usage trade-offs among stage, PHV, and memory. Rather than sending all of them to the constraint-based filter & optimizer module, we propose a heuristic algorithm that filters out strategies that violate basic constraints such as memory and stage.

In this approach, we only focus on comparing two metrics: stage saving and memory overhead. §6 would take more resources into account. If a strategy's memory overhead takes more stages than it can save by removing dependencies, it must end up occupying more stages than the original program, which conflicts with our goal. To sum up, given a P4 program, our heuristic algorithm runs as follows:

- Given a P4 program P, we generate its match-action DAG, D_P, and find all pairs of tables that potentially could be merged according to any of our primitives (mentioned in §5.1). Suppose we find n pairs.

- We build a binary decision tree T with n layers. Each layer represents one pair of tables, and each branch presents whether we remove the dependency of this pair of tables or not. Thus, a path from the root node of T to some leaf node of T represents a combination of table merging strategies.

- We thus run a deep-first search on T. During the searching process, we cut off the branches that violate basic memory and stage constraints. For each leaf node, we compute $S_{save} * m > M$, where S_{save} is the number of stages this strategy saves, m is the memory space of a single stage, and M is the memory overhead this strategy actually introduces. Note that S_{save} and M are computed by our primitives. If $S_{save} * m > M$, we keep this leaf node as one of our candidates used as the input of constraint-based filter & optimizer module (§6); otherwise, we drop this strategy.

6 Constraint-Based Filter & Optimizer

This module takes as input all candidates generated by §5, and then encodes each candidate program with all hardware resource size and constraints (stored in Cetus's backend DB) into an SMT formula. Then, we call an SMT solver (*e.g.*, Z3 [7]) to synthesize a table location plan that uses the least memory and stage resources. Finally, we realize this plan in a P4 program that specifies the locations of tables via `pragma` instructions.

The key challenge is how to efficiently solve these SMT formulas (each representing a candidate with all constraints). We found that the existing encoding approaches (*e.g.*, Lyra [10]) may result in state explosion, because a great number of diverse hardware resources create a huge search space that exceeds the SMT solver's searching capability.

To address the above challenge, we introduce a new approach that contributes two novel designs: (1) a new PHV encoding approach that significantly reduces the size of SMT formulas to avoid state explosion problem (§6.1); and (2) a two-step solving algorithm that decouples the solving process into table-related resource and variable-related resource solving to speed up the solving process (§6.2).
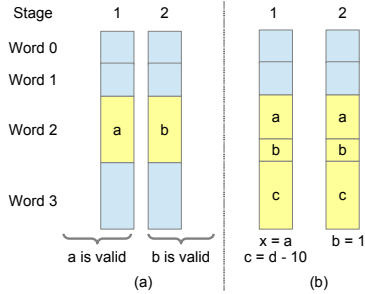
Figure 8: PHV sharing (a) across stages, (b) in one stage.

6.1 PHV Sharing Encoding

Packet Header Vector (PHV) serves as the bus between stages. The basic component of PHV is called word. There are tens of words with 8, 16, and 32-bit width respectively. One field can occupy one or multiple words. For example, a 48-bit source MAC field can take one 32b and one 16b word or three 16b words.

PHV is a scarce resource and needs careful planning, especially when the program is large and involves lots of headers and metadata. Simply adopting encoding approaches (*e.g.*, Lyra [10]) would waste the precious PHV spaces and fail to find a feasible solution. This is because Lyra's encoding assumes each word is dedicated to one variable; however, PHV words can be shared across variables in the PISA architecture, both across stages and within the same stage.

PHV sharing across stages. Different variables can occupy the same word at different stages. As shown in Figure 8(a), after stage 2, variable a is no longer used and another variable b can take over the same word. This allows us to use only one PHV container to store two independent variables that would otherwise require two containers. This sharing requires the variables have non-overlapping lifetimes, *i.e.* from the stage they are created till the last stage they are used. Note that all packet header fields' lifetime is the entire pipeline since they are created by the parser and consumed by the deparser. So the cross-stage sharing only applies to the metadata.

PHV sharing within one stage. Variables can also share the same word in the same stage as long as this sharing does not affect the correctness. Shown in Figure 8(b), variable a is read in stage 1 and variable b is assigned to a new value in stage 2. These two variables can share the same word. But variable c can not share with a at stage 1 because it was written by a subtract instruction. This is constrained by the fact that the Arithmetic Logic Unit (ALU) can perform at most one instruction to one word in one stage. The same-stage sharing applies to both header fields and metadata.

Cross-stage and same-stage sharing pack more variables into PHV, and it poses great pressure on PHV encoding. Because of the cross-stage sharing, we have to encode each stage's PHV allocation separately. The same-stage sharing further complicates the problem since we need to consider whether each pair of variables could share the same word.
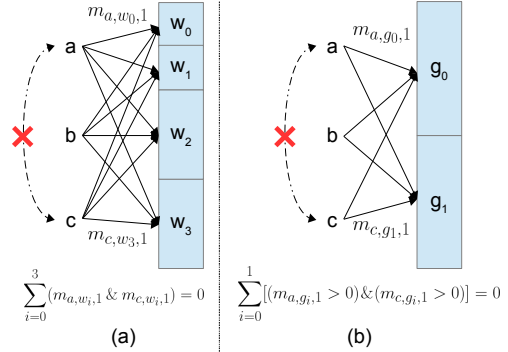

Figure 9: PHV encoding for 3 variables and 4 PHV words. (a) Strawman solution introduces 12 mapping variables and 4 rules. (b) Our solution reduces it to 6 mapping variables and 2 rules.

A strawman solution. A strawman solution is to encode the mapping $m_{v,w,s}$ between the variable v and each PHV word w at stage s. It encodes the cross-stage sharing by treating each stage separately. As for same-stage sharing, when two variables v_1 and v_2 cannot share the same word, we can add the constraint $m_{v_1,w,s}$ & $m_{v_2,w,s} = 0$. Next, we encode constraints such as each word has its own size limit, each variable should reserve enough bits in the PHV, *etc.* However, shown in Figure 9(a), because there are tens of stages and hundreds of PHV words, this solution introduces too many such mapping variables and the search space is huge.

Our encoding. Our PHV sharing encoding method addresses the scalability challenge. We observe that the total number of independent mappings in the encoded formula is the key complexity contributor. Thus, our focus is to reduce the independent mappings.

For same-stage sharing, we remove the boundary between PHV words and focus on whether variables can share with each other. We noticed that at each stage, there are only a few "shareable groups", the set of variables that can share with each other. Note that one variable can belong to multiple groups since the shareability is not transitive, *i.e.* variable v_1 can share with v_2 and v_3 cannot conclude v_2 can share with v_3. Then we can maintain the mapping between variables and these groups instead of the PHV words and restore PHV mapping afterward.

We also observe that in the encoded formula, all groups are symmetric: it does not affect the correctness when we reorder the groups. This is also another slow-down factor since it gives the SMT solver more freedom. To break the symmetry, we give preference to the groups with lower ID, the SMT solver can only use a new group until all the groups with lower ID are already assigned.

To summarize, the PHV encoding works as follows:

- (1) Given the input program $\mathcal{P}$, we count total number of non-assignment instructions I in each pipeline. This is the upper bound of the number of groups.

- (2) (Cross-stage sharing) For each variable v, we maintain: i) the mapping $m_{v,g,s}$, which denotes the number of bits v assign to group g at stage s, ii) the lifecycle l_v and r_v, which denotes the start and end stage of v.

- (4) (Same-stage sharing) If v_1 cannot share with v_2, then $(m_{v_1,g,s} > 0)$ & $(m_{v_2,g,s} > 0)$ is always false.

- (5) (Variable width) For each variable, if stage s is within the its lifecycle, the total number of bits in each group equals variable width b_v: $l_s \le s \le r_s \rightarrow \sum_i m_{v,g_i,s} = b_v$. Otherwise the summation is 0.

- (6) (PHV size) The summation of total number of bytes in each group should be less than PHV size. $\sum_i \lceil \sum_j m_{v_j,g_i,s}/8 \rceil \le N_{PHV}$.

- (7) (Break symmetry) We prioritize groups with lower ID: $(\sum_j m_{v_j,g_{i+1},s}) > 0 \rightarrow (\sum_j m_{v_j,g_i,s}) > 0$.

In Figure 9(b), because only a cannot share with c, there are at most 2 shareable groups. We introduce 2 groups g_0 and g_1. Through this encoding, we can reduce the total mapping from 12 to 6. In reality, there is at least one order of magnitude fewer groups than the PHV words. This can greatly reduce the encoded formula's complexity.

6.2 Two-Step Solving

The PHV sharing encoding optimization can greatly reduce the encoded formula's complexity, but the SMT solver still struggles when dealing with large-scale production programs. Due to their scale, the encoded formula is still too complex. Additionally, PISA architecture's table-related resources (*i.e.* memory, table stage) and variable-related resources (*i.e.* PHV, crossbar) are orthogonal to each other: how much memory the table allocates per stage does not affect where the variable is located in the PHV. This loose coupling relationship forms a huge search space and exceeds SMT solver's searching capability under large scale programs.

While this loose coupling is the culprit, it offers us an optimization opportunity. We can safely ignore their correlation and split the SMT solving problem into two smaller problems. The two-step solving works as follows:

- Given a P4 program (*i.e.*, one of the candidates), we encode all table-related resources and constraints and find a feasible plan P_t meeting dependency and constraints.

- Upon P_t, we encode variable-related resources and constraints, and call the SMT solver to find a solution P_v capable of meeting resources (*e.g.*, PHV and crossbar) and constraints.

- If yes, with P_t and P_v, we have $P = P_t + P_v$ as a resource allocation plan for the input P4 program, returning plan P.

- If not, we return to step 1, find another feasible plan P_t', P_v'.

- We repeat the above process until we find a valid plan P; otherwise, there is no valid plan for the input program.

This two-step approach can greatly improve the efficiency of our SMT solving. This aligns with our previous findings in §3.1 that the allocation of stages and table is our major concern. Other resources still remain and are more flexible.

6.3 The Best Result Selection

At the end of our workflow, the constraint-based filter & optimizer module may output one or more results that meet all already-known resource size and constraints. We select the most optimal one based on our internally-defined metric calculator. However, our experience shows that the constraint-based filter & optimizer module returns only one result in most cases.

7 Control Plane APIs Converter

After P' is obtained, our last task is to synthesize a control plane converter, making sure that the control plane APIs generated from the original program P are compatible with P' without any modification. Although different dependency removal primitives require different converting strategies, they follow the same underlying principle: generate new table entries that replace the previous tables' dependencies.

Due to limited space, we briefly describe the API converter for a concrete case shown in Figure 7(d) when installing new table entries. The rest of cases are detailed in Appendix A.

Let t_1, t_2 be the tables match c and e in program P, and t_1', t_2' be the tables after processing. In this example, t_1' is the same as t_1. In the runtime, the converter keeps a record of existing entries in table t_1 and t_2 installed from the control plane.

When inserting an entry e_1 to table t_1, we first insert e_1 into table t_1' unmodified. Next, for each existing entry e_{2i} in table t_2, create two new entries, one hits both e_{2i} and e_1, action is $b = p\_a$; one matches e_{2i} but misses e_1, action is $b = a\_0$. Insert all of them into table t_2'.

When inserting an entry e_2 to table t_2, for each existing entry e_{1i} in table t_1, we create two new entries as well. If table t_1 is empty, only create one rule that matches e_2 and other fields left wildcard. Other operations such as modifying or deleting an entry follow the same principle.

8 Deployment Experience

Cetus has been used to facilitate the development of P4 programs at Alibaba for one year. It has effectively decreased our P4 development workload by two orders of magnitude (from $O(\text{day})$ to $O(\text{min})$) This section presents several real cases addressed by Cetus.

Case 1: Parallelizing network functions. A common problem our programmers frequently encountered is that implicit dependencies between actions or hardware constraints may prevent two or multiple network functions from occupying the same stages. If one of the functions contains a large table and another function consists of multiple small tables forming a long dependency chain, the total number of occupied stages could exceed the number of stages available, and our programmers had no clue on how to fix such a problem.

(a) The original layout
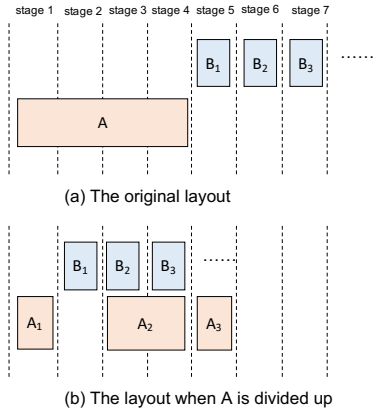

(b) The layout when A is divided up

Figure 10: Parallelizing network functions via Cetus.

Figure 10 shows a real case in our edge gateway program. In the original P4 program, network function A only has one table A, which is a large table for load balancing. Function B consists of multiple tables like B_1, B_2, B_3, *etc.* formulating a chain of small tables, each of which being responsible for inserting customized metadata for diverse services. However, if the program places network functions A and B as shown in Figure 10(a), a fitting issue occurs because their resource usage exceeds total stages available. From the view of our programmers, they can only do trial and error.

Through the dependency removal algorithm introduced in Section 5, Cetus can automatically address this problem by parallelizing network functions within few minutes. As shown in Figure 10(b), Cetus detected there is a deep dependency between actions of A_1 and B_1, thus dividing function A into a few tables and maximizing the parallelization of table placement. We used the solution in [23] to guarantee the split tables act the same as the original one.

Case 2: Optimizing write-after-write dependency. Using global data is common in many programming languages and software systems. However, such practice comes with pitfalls in P4 programs. For instance, because the physical pipeline offers control registers, our programmers are allowed to explicitly drop a packet in packet validation, access control, and error handling. However, write operations to a common field issued by different modules may constitute write-after-write dependencies, which cause the number of required stages to exceed the actual stage number.

Figure 11 shows a real case. Figure 11(a) is the original P4 program. Two tables are invoked consecutively, which may call the same action to explicitly drop the packet. Because of write-after-write dependency, they must occupy two stages. Due to the "lengthy diameter" feature in our production programs, a fitting issue happened because stage resources are overly used. We therefore called Cetus to solve our fitting issue. Cetus automatically generates a program shown in Figure 11(b). We can observe that the two tables in the original program are merged into one, saving one stage to enable the program to compile. More interestingly, Cetus can also care-

```
action drop_packet(){
  eg_dprsr_md.drop_ctl = 1;
}

table color_drop(){
  key = { meta.pkt_color: exact; }
  actions = { drop_packet; NoAction; }
}

table mirror_drop(){
  key = { meta.pkt_color: exact;
          meta.mirror: exact }
  actions = { drop_packet; NoAction; }
}

control(){
  color_drop();
  mirror_drop();
}
```

```
action drop_packet(){
  eg_dprsr_md.drop_ctl = 1;
}

table color_mirror_drop(){
  key = { meta.pkt_color: exact;
          meta.mirror: ternary }
  actions = { drop_packet; NoAction; }
}

control(){
  color_mirror_drop();
}
```

(a) write-after-write dependency that requires two stages   (b) merged tables that require only one stage

Figure 11: Write-after-write optimization

```
action set_flow_tag(bit<16> tag){
  meta.tag = tag;
}

table color_flow(){
  key = { meta.ingress_port: exact; }
  actions = { set_flow_tag; NoAction; }
}

action set_sample_rate(bit<16> rate){
  meta.rate = rate;
}

table sample_rate(){
  key = { meta.tag: exact; }
  actions = { set_sample_rate;
              NoAction; }
}

control(){
  color_flow();
  sample_rate();
}
```

```
action set_tag_rate(bit<16> tag,
                    bit<16> rate){
  meta.tag = tag;
  meta.rate = rate;
}

table generated_tbl(){
  key = { meta.ingress_port: exact; }
  actions = { set_tag_rate; NoAction; }
}

control(){
  generated_tbl();
}
```

(a) read-after-write dependency that requires 2 stages   (b) merged tables that require only one stage

Figure 12: Read-after-write optimization

fully merge the match keys from the two tables. Because the `color_drop` table does not match `meta.mirror` so the merged table used ternary to match `meta.mirror`.

Case 3: Optimizing read-after-write dependency. Modularization is another common paradigm in program development. By clearly defining interfaces and decoupling modules, it allows the independent design and development of individual pieces of code. However, the modularization of P4 programs often comes at the expense of RAW dependencies.

In our production P4 programs, it is common for one module to set a particular field, which is later read by another module. Figure 12(a) shows a real program example where the table `color_flow` tags each packet depending on which port it comes from. Then, another `sample_rate` table sets the sampling rate based on a packet's tag. This constitutes read-after-write dependency; thus, `sample_rate` has to be placed at least one stage later than `color_flow`, resulting in at least two stages occupied. We found such read-after-write dependencies are quite annoying in our programs because many fitting issues were caused by this type of dependency.

With Cetus in hand, we directly applied Cetus in this scenario. Cetus automatically analyzes whether it is better to trade-off modularization for more efficient and compact code, given the limited number of physical stages in each pipeline. In particular, Cetus checks whether `meta.tag` is solely determined by `color_flow`, and whether they are applied consecutively. If so, it merges the two tables so that the first and second lookup are performed simultaneously within one stage, as shown in Figure 12(b). As a side effect, merging these two

Program	LoC	Table Num (Ig/Eg)	Before		After		Dependency Removed			Time
			Diameter (Ig/Eg)	Stage Num	Diameter (Ig/Eg)	Stage Num	WAW	RAW	WAR	
PINT [3]	380	13 / 0	6 / 0	7	6 / 0	6	0	2	0	19s
RTT [16]	408	12 / 0	9 / 0	9	8 / 0	8	0	3	0	25s
Bier [18]	703	26 / 4	7 / 2	11	5 / 2	7	2	8	2	41s
P4_protect [17]	576	12 / 1	5 / 1	6	4 / 1	4	0	6	0	25s
Conquest [5]	847	1 / 19	1 / 7	9	1 / 6	6	0	8	0	2m51s
Beaucoup [6]	1677	25 / 0	10 / 0	12	10 / 0	11	0	1	0	6m58s
P4_switch	4701	34 / 25	8 / 5	12	8 / 5	11	0	2	0	11m30s
CDN	6342	19 / 2	10 / 2	11	10 / 1	10	0	3	0	1m27s
Edge vSwitch	2733	32 / 6	9 / 3	11	8 / 2	8	2	3	0	1m21s
Edge Gateway	4417	32 / 37	8 / 7	12	8 / 7	11	2	1	1	7m21s

Table 2: Experimental results conducted on a workstation with Intel Xeon 2.5GHZ CPU and 128GiB RAM

tables may cause the new table to occupy more memory; however, as designed in §6, Cetus is able to take both factors (*i.e.*, memory and stage) into account and produces a feasible solution if such optimization is indeed worthwhile.

Case 4: SDE upgrade. As the programs keep evolving, we also upgrade the runtime and development-time infrastructure, including the versions of switch OS and the P4 compiler, to enjoy the latest performance optimizations and fixes provided by the vendors. In such an upgrading case, the program must be re-fit. We can consult Cetus to pinpoint the problem and search for a feasible table layout. After being automatically annotated with `pragmas`, the existing P4 program was successfully compiled while keeping its code structure intact. In this way, Cetus cleared the most challenging obstacle and enabled the upgrade of the whole system.

9 Evaluation

Our evaluation aims to answer whether Cetus can reduce different program's stage usage (§9.1) and how effective the optimization algorithms are (§9.2). All experiments were performed on a server with 2.5GHz CPU and 768GiB RAM.

9.1 Optimization

We chose 10 P4 programs, 6 open-sourced and 4 private ones, to evaluate whether Cetus can optimize and reduce their stage usage. In this evaluation, we mainly show Cetus's stage occupation reduction capability. For each program, we record its DAG's diameter and the number of stages it occupies in ϒ chip before and after optimization. We further listed which types of dependencies Cetus removed and the time it took for each program. Table 2 shows the result.

First, Cetus removed 1 to 12 table dependencies, reduced the program's diameter by 1 to 2 and 1 to 4 stages. This shows the effectiveness of the primitives used by Cetus and our findings in §3.1 also apply to open source programs.

Second, Cetus can successfully find the best candidate at a decent speed. For simple programs, Cetus can find a plan in under a minute. For complicated ones, Cetus still managed to finish the search in minutes. Compared with the days of efforts developers spent optimizing the program manually, this is way faster and saves a lot of deployment efforts.

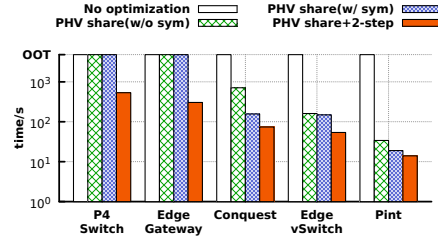Third, we can see that most of the dependencies removed

Figure 13: Time for a solution under different optimizations.

were RAW dependencies. This is because of two reasons: (1) RAW dependency is common in programs. (2) RAW dependency is hard to find and also hard to remove. For example, below is a code snippet from Beaucoup [6]:

```
if(ig_md.cf_key_matched==1) {
    exec_regcoupon_merge(); // writes coupon_merge_check
}
if (ig_md.cf_decay_has_expired==1) {
    exec_counter_set_to_one();
} else {
    if (ig_md.cf_key_matched==1 && ig_md.coupon_merge_check==0) {
        exec_counter_incr();
}}
```

In the above code, the action `exec_regcoupon_merge()` writes variable `coupon_merge_check`, which is later read by the condition of action `exec_counter_incr()`. Cetus removes their dependency through the RAW dependency removal primitive, and it reduces one stage occupation. But for developers, it is hard to notice because it is spread across two different condition branches far away.

9.2 Performance

To further evaluate the effectiveness of the optimization techniques introduced in §6, we chose several typical programs with different scales and run experiments with different optimization techniques enabled. Starting from the naive solution with no optimization, we add vanilla PHV sharing encoding, symmetry breaking encoding, and finally two-step solving to Cetus sequentially. We set 1 hour as the timeout threshold. The result is shown in Figure 13.

Without any optimization, all programs timed out, which means it is necessary to introduce optimizations. For small-scaled programs, such as Conquest, Edge vSwitch, and Pint, adopting PHV sharing encoding can greatly improve the performance, indicates that the bottleneck lines in the complexity of the encoded SMT formula. However, for large-scale pro-

grams, such as P4 Switch and Edge Gateway, we only met the deadline after adding all three optimizations. This shows that for large scale programs, encoding optimization is not enough, the search space is still too large for the SMT solver to handle. It is necessary to leverage the key findings in §3.1 and bring in two-step solving to give a hint to the SMT solver.

10 Discussion and Lessons

This section discusses our lessons and limitations.

Is P' functionally identical to P? In principle, Cetus's approaches, including table merging and constraint-based filter & optimizer, can only change and optimize the location of tables, rather than the function logic of programs; thus, P' should be functionally the same as P. While we have not manually proved our approach on this property, in Alibaba, we employ a P4 verification tool, Aquila [27], to check the consistency between P and P' when Cetus generates P'. If Aquila returns "yes", that means we can use P' to replace P. So far, we have not seen any inconsistency case.

Can Cetus capture all hardware constraints within Υ chip? We encode constraints as many as we can; thus, we can only make sure that P' will not violate any constraints we have encountered before. With the accumulation of more and more hardware constraints, we believe the capability of Cetus will become stronger. However, we cannot guarantee every P' can compile to Υ chip. We did experience few cases that P' does not compile due to unknown constraints.

Can lengthy diameter always hold? We cannot guarantee the lengthy diameter can always exist in our production programs in the future; however, based on our experience with Cetus so far, the stage shortage issue resulting from the lengthy diameter is still the highest priority barrier in our scenario. We thus suggest the ASIC vendor consider releasing a chip with double the number of stages and less memory.

Cetus's limitations. We have the following main limitations. First, Cetus can only remove dependencies like WAW, RAW, and WAR. Cetus cannot handle more tricky cases such as removing dependency via modifying program semantic. Both RAW dependency removal algorithms require a third table in front to parallelize the latter two tables. For programs such as Syncookies [22], Cheetah [29], because they have long, chained sequential computations, the requirement of RAW dependency removal is not met, Cetus cannot perform optimizations. Second, Cetus cannot optimize a program when it occupies too many resources, since the dependency removal algorithms come at the cost of additional resources in the switch, such as PHV and memory. Third, we cannot guarantee Cetus's implementation is bug-free although we spent a lot of time checking our implementation bugs; thus, sometimes the output P' may not be the best one. Finally, if a new programmable ASIC architecture is introduced, Cetus cannot be directly used to generate compilable programs for this new ASIC. Cetus has to encode all constraints of this new ASIC.

11 Related Work

P4 program optimizers and compilers. This type of systems optimize resource usage in programmable ASICs or simplify programmers' tasks on expressing their coding intent. P4All [13, 14] aims to optimize resource usage by leveraging reusable data structures, such as bloom filters and key-value stores; however, our production P4 programs do not share these data structures. P4visor [31, 32] optimizes resources by merging redundant code fragments (*e.g.*, header parser and tables). P4visor is a good complementary to Cetus. Before Cetus was developed, we already built an internal system (similar to P4visor) to merge redundant code fragment. μP4 [26] proposes a modular way to write P4 code. Jose *et al.* [15] compiles P4 programs to architectures such as the RMT and Flex-Pipe. Domino [24] and Lyra [10] simplify data plane programming by specifying C-like new languages. Chipmunk [11, 12] leverages slicing, a domain-specific synthesis technique, to remove unnecessary resources cost by Domino. P²GO [30] proposes an idea that reduces the allocated resources of a P4 program based on traffic trace profiling. However, it might be hard for us to deploy it in our environment, because if unexpected traffic turns up after the profiling, some function might be already pruned. Different from the state of the art (that keeps the original dependencies), Cetus optimizes resource usage by removing dependencies in P4 programs.

Network-wide configuration synthesis. Configuration synthesis work [4,8,9,19,21,28] offers the operator network-wide abstractions for configuration synthesis. SyNET [8] and ConfigAssure [19] offer general abstractions to synthesize the protocol configuration. Recent work [9]indicates that none of the above systems is scalable to cloud-scale networks. Propane [1, 2], Snowcap [21], and Jinjing [28] synthesize BGP, updating, and ACL configurations, respectively.

12 Conclusion

We have presented Cetus, the first system that releases the P4 programmers from frustrating trial and error compiling. Cetus can automatically convert an uncompilable P4 program into a functionally identical but compilable P4 program. We have been using Cetus in our production P4 program development for one year, and it has effectively decreased our P4 development workload by two orders of magnitude (from $O(\text{day})$ to $O(\text{min})$).

This work does not raise any ethical issues.

Acknowledgments

References

[1] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 328–341, 2016.

[2] Ryan Beckett, Ratul Mahajan, Todd D. Milstein, Jitendra Padhye, and David Walker. Network configuration synthesis with abstract topologies. In *38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[3] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. Pint: Probabilistic in-band network telemetry. In *Proceedings of the 2020 ACM SIGCOMM Conference*, pages 662–680, 2020.

[4] Eric Hayden Campbell, William T. Hallahan, Priya Srikumar, Carmelo Cascone, Jed Liu, Vignesh Ramamurthy, Hossein Hojjat, Ruzica Piskac, Robert Soulé, and Nate Foster. Avenir: Managing data plane diversity with control plane synthesis. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2021.

[5] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, Ori Rottenstreich, Steven A Monetti, and Tzuu-Yi Wang. Fine-grained queue measurement in the data plane. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, pages 15–29, 2019.

[6] Xiaoqi Chen, Shir Landau-Feibish, Mark Braverman, and Jennifer Rexford. Beaucoup: Answering many network traffic queries, one memory update at a time. In *Proceedings of the 2020 ACM SIGCOMM Conference*, pages 226–239, 2020.

[7] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *14th Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[8] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin T. Vechev. Network-wide configuration synthesis. In *29th International Conference on Computer Aided Verification (CAV)*, 2017.

[9] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin T. Vechev. NetComplete: Practical network-wide configuration synthesis with autocmpleteion. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2018.

[10] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics. In *Proceedings of the 2020 ACM SIGCOMM Conference*, pages 435–450, 2020.

[11] Xiangyu Gao, Taegyun Kim, Aatish Kishan Varma, Anirudh Sivaraman, and Srinivas Narayana. Autogenerating fast packet-processing code using program synthesis. In *18th ACM Workshop on Hot Topics in Networks (HotNets)*, 2019.

[12] Xiangyu Gao, Taegyun Kim, Michael D Wong, Divya Raghunathan, Aatish Kishan Varma, Pravein Govindan Kannan, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. Switch code generation using program synthesis. In *Proceedings of the 2020 ACM SIGCOMM Conference*, pages 44–61, 2020.

[13] Mary Hogan, Shir Landau Feibish, Mina Tahmasbi Arashloo, Jennifer Rexford, and David Walker. Modular switch programming under resource constraints. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2022.

[14] Mary Hogan, Shir Landau Feibish, Mina Tahmasbi Arashloo, Jennifer Rexford, David Walker, and Rob Harrison. Elastic switch programming with P4All. In *19th ACM Workshop on Hot Topics in Networks (HotNets)*, 2020.

[15] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. Compiling packet programs to reconfigurable switches. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

[16] Elie Kfoury, Jorge Crichigno, Elias Bou-Harb, and Gautam Srivastava. Dynamic router's buffer sizing using passive measurements and p4 programmable switches.

[17] Steffen Lindner, Daniel Merling, Marco Häberle, and Michael Menth. P4-protect: 1+ 1 path protection for p4. In *Proceedings of the 3rd P4 Workshop in Europe*, pages 21–27, 2020.

[18] Daniel Merling, Steffen Lindner, and Michael Menth. Hardware-based evaluation of scalable and resilient multicast with bier in p4. *IEEE Access*, 9:34500–34514, 2021.

[19] Sanjai Narain, Gary Levin, Sharad Malik, and Vikram Kaul. Declarative infrastructure configuration synthesis and debugging. *J. Network Syst. Manage.*, 16(3):235–258, 2008.

[20] Tian Pan, Nianbing Yu, Chenhao Jia, Jianwen Pi, Liang Xu, Yisong Qiao, Zhiguo Li, Kun Liu, Jie Lu, Jianyuan Lu, et al. Sailfish: accelerating cloud-scale multi-tenant multi-service gateways with programmable switches. In *Proceedings of the 2021 ACM SIGCOMM Conference*, pages 194–206, 2021.

[21] Tibor Schneider, Rüdiger Birkner, and Laurent Vanbever. Snowcap: synthesizing network-wide configuration updates. In *Proceedings of the 2021 ACM SIGCOMM Conference*, pages 33–49, 2021.

[22] Dominik Scholz, Sebastian Gallenmüller, Henning Stubbe, Bassam Jaber, Minoo Rouhi, and Georg Carle. Me love (syn-) cookies: Syn flood mitigation in programmable data planes. *arXiv preprint arXiv:2003.03221*, 2020.

[23] Devavrat Shah and Pankaj Gupta. Fast updating algorithms for tcam. *IEEE Micro*, 21(1):36–47, 2001.

[24] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 15–28, 2016.

[25] John Sonchack, Devon Loehr, Jennifer Rexford, and David Walker. Lucid: A language for control in the data plane. In *Proceedings of the 2021 ACM SIGCOMM Conference*, pages 731–747, 2021.

[26] Hardik Soni, Myriana Rifai, Praveen Kumar, Ryan Doenges, and Nate Foster. Composing dataplane programs with μp4. In *Proceedings of the 2020 ACM SIGCOMM Conference*, pages 329–343, 2020.

[27] Bingchuan Tian, Jiaqi Gao, Mengqi Liu, Ennan Zhai, Yanqing Chen, Yu Zhou, Li Dai, Feng Yan, Mengjing Ma, Ming Tang, et al. Aquila: a practically usable verification system for production-scale programmable data planes. In *Proceedings of the 2021 ACM SIGCOMM Conference*, pages 17–32, 2021.

[28] Bingchuan Tian, Xinyi Zhang, Ennan Zhai, Hongqiang Harry Liu, Qiaobo Ye, Chunsheng Wang, Xin Wu, Zhiming Ji, Yihong Sang, Ming Zhang, et al. Safely and automatically updating in-network acl configurations with intent language. In *Proceedings of the 2019 ACM SIGCOMM Conference*, pages 214–226. 2019.

[29] Muhammad Tirmazi, Ran Ben Basat, Jiaqi Gao, and Minlan Yu. Cheetah: Accelerating database queries with switch pruning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 2407–2422, New York, NY, USA, 2020. Association for Computing Machinery.

[30] Patrick Wintermeyer, Maria Apostolaki, Alexander Dietmüller, and Laurent Vanbever. P2GO: P4 profile-guided optimizations. In *The 19th ACM Workshop on Hot Topics in Networks (HotNets)*, 2020.

[31] Peng Zheng, Theophilus Benson, and Chengchen Hu. P4visor: Lightweight virtualization and composition primitives for building and testing modular programs. In *14th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2018.

[32] Peng Zheng, Theophilus A. Benson, and Chengchen Hu. Building and testing modular programs for programmable data planes. *IEEE J. Sel. Areas Commun.*, 38(7):1432–1447, 2020.
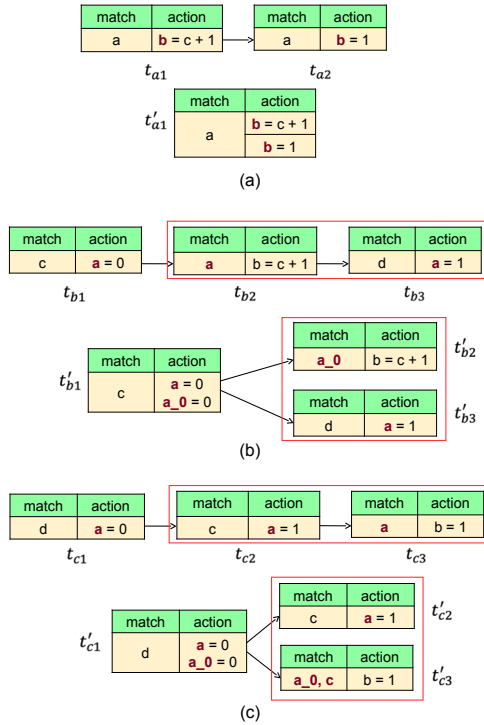
Figure 14: Examples for control plane APIs converter: (a) WAW (b) WAR (c) RAW-match.

APPENDIX

Appendices are supporting material that has not been peer-reviewed.

A Control Plane APIs Converter

This section details how Cetus's control plane API converter bridges the inconsistency between the original program P and the optimized one P'. We labeled the tables in Figure 7 and show the example tables in Figure 14.

Write-after-write dependency. Since two tables t_{a1} and t_{a2} in Figure 14(a) share the same match field, the entries for both tables are inserted to the merged table t_a' directly. However, when two entries e_1, e_2 for t_{a1} and t_{a2} respectively overlaps their match field (*e.g.* e_1 matches 10.0.0.0/8 while e_2 matches 10.0.0.0/16), entry e_2 has higher priority than e_1 because table t_{a2} applies later than t_{a1}.

Write-after-read dependency. The match field of table t_{b2} is renamed. For an entry e_2 inserted to table t_{b2}, Cetus renames the match fields' name and inserts it to table t_{b2}'. For example, in Figure 14(b), the match field a in e_2 is renamed to a_0. Entries for table t_{b3} are inserted to table t_{b3}' directly.

Read-after-write-match dependency. In this case, Cetus records all the entries inserted to table t_{c2} and t_{c3} in a 'logical table' stored in memory. When a control plane application

inserts an entry e_2 to table t_{c2} with match value c_{e_2}, Cetus first inserts e_2 to table t_{c2}' unmodified. Next, if there exists an entry recorded in logical table t_{c3} that matches the result of action in table t_{c2}, which is $a = 1$ in Figure 14(c), then Cetus creates a new entry e_2' that matches c with value c_{e_2} and ignores value of a_0 and inserts it to table t_{c3}'. When an entry e_3 is inserted to table t_{c3}, there are two cases. If e_3 matches the result of the action in table t_{c2}, record it in the 'logical table' and do not insert it anywhere. Otherwise, rename the match field name of e_3 from a to a_0, add another match field c in e_3 but ignores the value. The 'ignore' can be expressed by using the wildcard if t_{c3}' uses TCAM memory, or by enumerating all possible values if it uses SRAM memory.

Read-after-write-action dependency. This part has been detailed in §7.

The entry removal operation is the reverse of the above actions.

Exploiting Digital Micro-Mirror Devices for Ambient Light Communication

Talia Xu, Miguel Chávez Tapia, and Marco Zúñiga

Technical University Delft

{m.xu-2, m.a.chaveztapia, m.a.zunigazamalloa}@tudelft.nl

Abstract

There is a growing interest in exploiting *ambient light* for wireless communication. This new research area has two key advantages: it utilizes a free portion of the spectrum and does not require modifications of the lighting infrastructure. Most existing designs, however, rely on a single type of optical surface at the transmitter: liquid crystal shutters (LCs). LCs have two inherent limitations, they cut the optical power in half, which affects the range; and they have slow time responses, which affects the data rate. We take a step back to provide a new perspective for ambient light communication with two novel contributions. First, we propose an optical model to understand the fundamental limits and opportunities of ambient light communication. Second, based on the insights of our analytical model, we build a novel platform, dubbed PhotoLink, that exploits a different type of optical surface: digital micro-mirror devices (DMDs). Considering the same scenario in terms of surface area and ambient light conditions, we benchmark the performance of PhotoLink using two types of receivers, one optimized for LCs and the other for DMDs. In both cases, PhotoLink outperforms the data rate of equivalent LC-transmitters by factors of 30 and 80: 30 kbps & 80 kbps vs. 1 kbps, while consuming less than 50 mW. Even when compared to a more sophisticated multi-cell LC platform, which has a surface area that is 500 times bigger than ours, PhotoLink's data rate is 10-fold: 80 kbps vs. 8 kbps. To the best of our knowledge this is the first work providing an optical model for ambient light communication and breaking the 10 kbps barrier for these types of links.

1 Introduction

In the last two decades, the adoption of wireless communication has gone through an unprecedented expansion. This ever-increasing demand has raised warnings of a looming 'radio frequency (RF) crisis' [5], and various alternative technologies are being explored to mitigate this risk. Among them, visible light communication (VLC) has gained significant attention due to its wide, free and unregulated spectrum. VLC is a sub-area of optical wireless communication (OWC) that focuses on light sources that are incoherent, divergent and multichromatic (such as sunlight and artificial white light). VLC allows standard LEDs to provide illumination and communication and it is enabling several novel applications, from interactive toys [23], indoor positioning systems [27], to LiFi [20]. VLC, however, has an important limitation: it requires direct (*active*) control over the circuitry of the light source to modulate its intensity. Most of the light in our environments comes from sources we cannot control directly, not only the sun but also plenty of artificial lighting.

To exploit the vast presence of *ambient light*, researchers are investigating backscattering (*passive*) communication. Passive-VLC modulates ambient light using liquid crystal shutters (LCs). LCs can be seen as light shutters that allow (or block) the passage of light to communicate logical ones (or zeros). Recent studies report ambient light links reaching more than 50 m with data rates around 1 kbps, while consuming only a few mWs [7, 25]. Ambient light communication is a transformative eco-friendly concept because it piggybacks on top of energy that already exists, but current passive-VLC studies face two main challenges.

Challenge 1*: There has been no optical analysis of various passive VLC systems.* In a way, our community has rushed into the design of systems without carrying out first a proper optical analysis of the various types of ambient light and their impact on communication. Hence, several designs have been implemented reporting a wide range of *(i)* coverages (from a few meters to several tens of meters), *(ii)* data rates (from hundreds of bps to several kbps), and *(iii)* lighting conditions (from cloudy and sunny days to various types of artificial lighting). However, without an analytical framework, it is difficult to define a common baseline to directly compare and understand which elements contribute to such disparate performance. More importantly, we cannot provide insights about the fundamental opportunities and limits of ambient light communication.

Challenge 2*: Transmitters focus on a single optical device.* State-of-the-art (SoA) designs in passive VLC studies have been mainly constrained to a single type of optical surface,

the LCs, but LCs have some inherent limitations. First, even before any type of modulation begins, LCs cut the optical power in half due to the use of polarizers. This undesirable, but necessary, property of LCs reduces the communication range. Second, LCs have inherently slow rise and fall times, which has limited the data rate of all *single-cell* designs to values around 1 kbps [7, 13, 29]. Our design space could broaden greatly if we include other types of optical surfaces.

In this work, we take a step back to rethink passive-VLC. First, we propose a simple optical model to gain fundamental insights. Then, based on the outcomes of our model, we explore the use of digital micromirror devices (DMDs), which have different operating principles compared to LCs. In particular, our work makes the following contributions:

Contribution 1 [section 2]*: An optical model for ambient light communication.* Our model includes a key optical principle that has not been considered in ambient light communication: the fact that the performance depends not only on the luminous flux of the light source (output power) but also on its radiation pattern (diffused or directional). For example, this insight explains why a system tested under artificial light can perform better than under diffuse sunlight, even though diffuse sunlight can provide illumination that is an order of magnitude higher than artificial lighting.

Contribution 2 [section 3]*: A new type of transmitter device.* Our model shows that maintaining directional light patterns is central for passive links, but maintaining such directionality requires the right type of *(i) ambient light* and *(ii) transmitter* (optical surfaces with specular reflection). To attain that goal, we propose a novel transmitter based on DMDs. Inexpensive DMDs, however, are designed for video projection and provide slow update rates, around a few hundred Hz. We design a custom controller to generate carriers up to 220 kHz. Our novel transmitter provides higher contrast and faster switching speed, allowing us to increase the data rate of passive links by a factor of 80 compared to LC transmitters.

Contribution 3 [section 4 and section 5]*: An implementation and thorough evaluation of our platform.* We build two transmitters, one with a DMD and the other with an LC; and two receivers, one optimized for LCs and the other for DMDs. Using the same setup for all evaluations, in terms of surface area and illumination, our results show that (i) if we use the receiver optimized for LCs, PhotoLink attains 30 kbps for a distance of six meters and a BER below 1%, compared to the 1 kbps provided by the LC for the same range and BER [3, 7, 29], (ii) if we use the receiver optimized for DMDs, the data rate increases to 80 kbps. This performance is obtained with a power consumption around 45 mW. Furthermore, even if we compare PhotoLink with a *multi-cell* LC system having a surface area that is 500+ times bigger than ours (66 cm^2 vs. 0.13 cm^2) [28], PhotoLink can achieve an order of magnitude higher data rate (80 kbps vs. 8 kbps). To the best of our knowledge, our work is the first to break the 10 kbps barrier with ambient light communication.

2   System Analysis

A passive VLC system has three basic components, the emitter (light source), the transmitter (modulating surface) and the receiver. Every SoA study adopts a different set of components. Some studies use a light bulb as the emitter, others use a flashlight or the sun. Some studies use a diffuser at the modulating surface, others use retro-reflectors or aluminium plates. Some studies use lenses at the receivers, others do not. This wide range of set ups is, in part, responsible for the equally wide range of performances reported in the literature, with data rates ranging from 0.5 kbps to 8.0 kbps to link distances ranging from 2 m to 80 m [3, 13, 25, 26, 28, 29].

Leaving aside the specific modulation methods of all these studies, we want to gain a fundamental understanding of passive systems and their components. Building upon the models developed for free-space optics [21], we propose a framework to analyze passive communication with ambient light.

2.1   Maintaining the luminous flux

First, let us start with a guideline that, to the best of our knowledge, has not been stated in any prior passive-VLC study: *The most important aim in passive communication is to convey as much* LUMINOUS FLUX *as possible from the emitter to the receiver.* The *luminous flux*, which is measured in *lumen*, is different from *illuminance*, which is measured in *lux* (lux = lumen per unit area). To compare two different systems fairly, one should know at least the area and the illuminance at the transmitter (modulating surface). This represents the amount of energy that is captured by the transmitter (E_C). Unfortunately, few studies report these two pieces of information.

The luminous flux, however, is not the only important parameter. Equally important is the radiation pattern, which determines how much luminous flux is maintained throughout the optical link (i.e., how much of E_C is able to arrive at the receiver). To highlight the importance of the radiation pattern, Fig. 2 depicts a *specular (mirror-like)* surface under four different types of light sources. The effect on the luminous flux is shown from more to less directive:

a) Ideal. First, to exemplify an ideal setup, let us use a laser, which is a highly directional source where the luminous flux is hardly lost. Due to this property, lasers are used extensively for long-distance free-space communication. Lasers, however, are a fundamentally different type of light source that is not as pervasive (or safe) as natural or artificial white light, and therefore, it is considered only as a reference in this paper.

b) Directional (sunlight in a clear day). On a clear day, sunlight rays travel in parallel and a specular surface maintains that directionality (luminous flux) towards the receiver. *We found only one study exploiting this setup, but with LCs [7]. Our platform shows the significant gains that can be obtained in this setup using DMDs.*

c) Lambertian (light bulbs and flashligths). With light bulbs, only a fraction of the luminous flux radiated by the source reaches the surface (green arrows in Fig. 1c). Further-

(a) Ideal
[laser]

(b) Directional
[direct sunlight]

(c) Lambertian
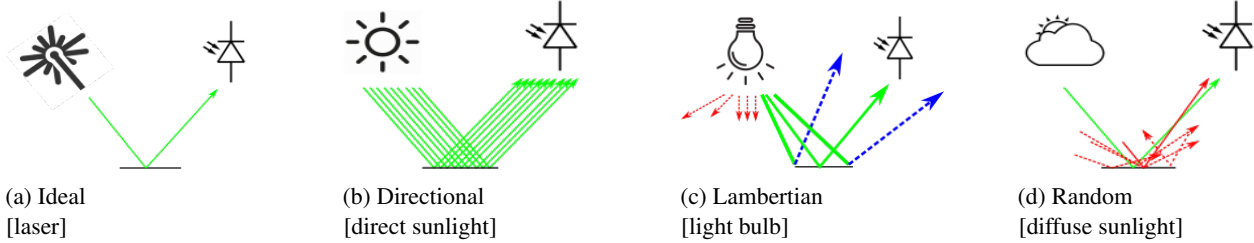[light bulb]

(d) Random
[diffuse sunlight]

Figure 1: The effect of different radiation patterns on the luminous flux. The reflective surface is specular.

more, since rays are radiated in different angles, when the luminous flux hits the surface, some rays are lost because the impinging angle is either too broad or too narrow to hit the receiver (blue arrows). *This scenario is used by all the backscattering studies reported in the literature [13, 25, 28, 29].*

d) Random (sunlight in a cloudy day). Clouds scatter sunlight, emitting rays uniformly in random directions. Due to this phenomenon, only an infinitesimally small fraction of the rays will impinge the surface at the right angle to reach the receiver (green arrow in Fig. 1d). *Our model shows that this is the worst case scenario with specular surfaces. No practical links can be obtained in this setup.*

The key point of this preliminary analysis is to highlight the importance of maintaining the luminous flux throughout the optical link. In the next subsection, we present a model to capture more detailed insights with a ray-tracing simulator.

2.2 Ray-tracing model

A 2D representation of a typical passive system is shown in Fig. 2a. The optical link has two main parts. *First*, the link between emitter and transmitter. Light is emitted from the light source O_L, with a (yellow) wavefront represented by A_L. The modulating surface O_T, acting as a transmitter, is at a distance D_{LT} from the light source, and receives a fraction of the luminous flux emitted by O_L. *Second*, the link between transmitter and receiver. The flux reflected by the surface O_T is represented with a (blue) wavefront A_T [1]. The photoreceiver O_R is at a distance D_{TR} from the transmitter, and collects only a fraction of the flux reflected by O_T. Another relevant parameter is the Field-of-View (FoV) of the receiver, which is represented by α_R (purple coverage). A wide FoV can cope with movements at the transmitter, but captures more noise.

Our toolbox, based on the above described model, is built upon Optometrika, a ray-tracing tool [18]. In essence, the toolbox divides the surface of the emitter, transmitter and receiver into small elements and calculates the fraction of rays that are able to reach the receiver. To assign the correct weight to each ray, Optometrika considers important optical parameters such as the angles of radiation, incidence and reflection. To analyze ambient light communication, the key inputs we need to provide to the toolbox are the radiation patterns of the emitter and the modulating surface.

[1]It is important to note that our model also captures the performance of retroreflectors because, from an optical perspective, the reflected radiation patterns are similar to those caused by mirrors
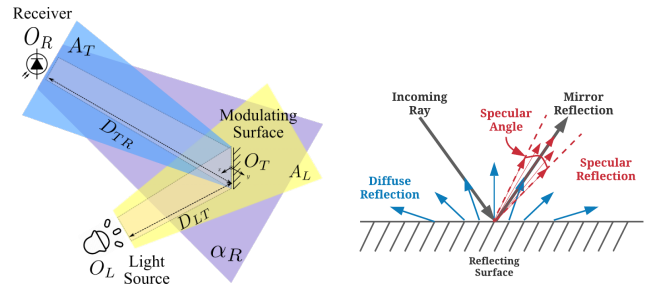
(a) A 2D representation of the optical system

(b) Different types of reflections based on the Phong model [19].

Figure 2: Optical system and different reflection types.

2.3 Insights & Guidelines

A passive link is, in essence, a triplet <emitter, transmitter, receiver> that finetunes the parameters of each element to optimize the performance. To analyze the complete design space, including the systems proposed in prior studies, we utilize a few abstractions for the emitters and transmitters, as presented in Tables 1 and 2.

Unless indicated otherwise, our analysis assumes that (i) there is no noise, which is similar to conducting experiments in the dark, (ii) the illuminance on the transmitter is fixed at 1800 lx, to provide a common baseline for all cases and *remove the trivial case where the performance is increased by increasing the illuminance*, and (iii) the area of the receiver is 1×1 cm^2. The selected area has no real impact on the analysis. The only assumption we make is that the transmitter's area is bigger than the receiver's, which is the case for most systems. Also, for our initial analysis, the receiver's FoV does not play a role because we assume a dark environment. In practice, the FoV plays a critical role and we will discuss it later on.

Regarding the modulating surface, we consider two main reflective patterns, as shown in Fig. 2b: *diffuse reflection*, caused by rough surfaces that reflect light in all directions, and *specular reflection*, caused by smooth surfaces. We further classify specular surfaces based on their specular angle. If the angle is zero, we call it mirror reflection.

2.3.1 Choosing the right emitter and transmitter

The design space of passive links can be divided into six main blocks based on the <emitter, transmitter> pair. Table 3 shows previous works categorized in this manner. Considering that direct sunlight provides tens of thousands of lx, overcast sunlight thousands of lx and light bulbs only hundreds of lx, a

Table 1: Emitters

Source	Type	Size of O_L	D_{LT}
L1	LED	5 cm × 5 cm	1 m
L2	LED	5 cm × 5 cm	4 m
L3	Diffuse Sunlight	N/A	N/A
L4	Direct Sunlight	N/A	N/A

Table 2: Transmitters

Modulating Surface	Type	Specular Angle	Size of O_T	Illuminance
T1	Diffuse	N/A	3 cm × 3 cm	1800 lx
T2	Specular	0.3°	3 cm × 3 cm	1800 lx
T3	Specular	1°	3 cm × 3 cm	1800 lx
T4	Specular	5°	3 cm × 3 cm	1800 lx

Table 3: A taxonomy of passive VLC systems

Light Source \ Surface Type	Specular (includes retro-reflectors)	Diffuse
LED	RetroVLC [13] PassiveVLC [29] RetroTurbo [28] RetroI2V [25]	
Sunlight (Cloudy Day)		Tweeting with Sunlight (TwSL) [4]
Sunlight (Clear Day)	ChromaLux [7]	Luxlink [3]

designer may assume that for any given surface, sunlight will always perform better than light bulbs. Similarly, considering that specular (mirror) surfaces provide stronger reflections than diffuse surfaces, a designer may assume that for any type of ambient light, a specular reflector will always perform better. Neither assumption is correct. In fact, we show that a particular combination of sunlight and specular reflectors gives the worst performance.

Fig. 3 depicts the signal strength of various scenarios as a function of the transmitter-receiver distance (D_{TR}). We consider all six possible combinations of *emitters*: LED (L1 & L2), overcast day (L3), clear day (L4); and *transmitters*: diffuse (T1), specular (T2). Our results show four design regions, which are described next from worst to best. Our evaluation section validates many of these results empirically.

Region 1: cloudy day & specular surface (L3-T2 in Fig. 3a, gray area in Table 3). This region captures the scenario in Fig. 1d, where light arrives in a scattered manner and only an infinitesimal amount of the flux reaches the receiver. The signal strength of this setup is so weak and decays so fast, compared to the other scenarios, that it is not shown within the range of Fig. 3a to have a clearer view of the other regions.

Region 2: any light & diffuse surface (LX-T1, blue area). When a diffuse surface is used, it does not matter the radiation pattern of the light source, so long as the luminous flux at the transmitter's surface is the same. Note that all T1 curves overlap with each other in Fig. 3a. This occurs because ideal diffusers, such as paper or plaster, distribute the reflections of the impinging flux in all directions.

Region 3: LED & specular surface (L1/L2-T2, red area). This is the second best region, and coincidentally, the main focus of prior work using retro-reflectors. Artificial lights, however, offer a wide range of radiation patterns, resulting in widely different performance. To illustrate this point we use Fig. 3b, where two emitters are placed at 1 m and 4 m (L1 & L2). *Both emitters attain the same illuminance at the receiver* (1800 lx, a white light illuminance of 1800 lx over an 1m^2 surface is approximately equivalent to the power of a 25 W LED), but L2, which is *further away*, provides a stronger signal strength, which is counter-intuitive. This

occurs because the further away the light source is, the more it behaves as a distant point source, leading to more directional beams impinging on the transmitter, and hence, less flux lost towards the receiver, c.f. Fig. 1c. In practice, L1 could be seen as a light bulb and L2 as a flashlight, which explains why studies using a flashlight attain better results [25, 28].

Region 4: clear day & specular surface (L4-T2, green area). This is the best operation region. Note that the signal strength hardly decays in Fig. 3a. This occurs because the high directionality of clear sunlight maintains the luminous flux over long distances, which is why heliographs (mirrors) used in the 1800's reached ranges beyond 100 km. This same property can increase the data rate of ambient light links. In practice, air attenuates the signal strength (similar to what happens with lasers), but the benefits of directionality remain strong.

2.3.2 Choosing the right specular surface

The above analysis highlights the importance of maintaining directionality throughout the optical link. However, given that there are no perfect mirror-reflectors, how critical is the specular angle? A wide specular angle can be the result of imperfections on the surface. For example, many studies use retro-reflectors, but the quality of retro-reflectors can vary. Fig. 3c shows the signal strength of surfaces with different specular angles, from narrow (T2, 0.3°) to wide (T4, 5.0°), considering an LED (L1) and direct sunlight (L3). When an LED is used (blue lines), the misaligned radiation pattern of the LED is more relevant than the specular angle, therefore, there is not much difference among the various surfaces. However, for a directional source (red lines), a large specular angle (e.g. 5° for T4) can lead to a significant decrease in the signal strength. Thus, *the more directional the rays, the more critical is the use of high-quality specular surfaces.*

2.3.3 Choosing the right receiver

Passive-VLC systems use cameras and photodiodes as receivers. Cameras are widely available in smartphones, but they are power hungry and slow, allowing only a few hundred frames per second. Photodiodes (PDs), on the other hand, are inexpensive, low-power and have a high bandwidth. Thus,
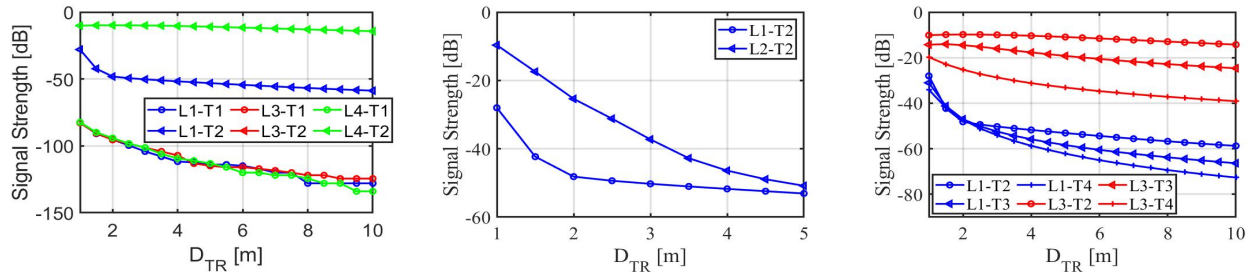
(a) Signal strength for different light source and surface combinations

(b) Signal strength for LEDs at different distances D_{LT}.

(c) Signal strength for different specular angles.

Figure 3: Different simulation setups.

PDs are the preferred choice for high data rate links. A key element in the PD's design is its FoV. The FoV will not only capture the intended signal but the surrounding noise as well (purple coverage in Fig. 2a). In practice, to maximize the SNR, the receiver's FoV should cover only the modulating surface, but that is difficult to attain. PDs with varying FoV have been used in the literature, ranging from $1°$ to close to $90°$ [3, 26]. Many studies using the wide FoV, however, were conducted at night with no interfering ambient light, which is similar to having a nearly perfect FoV of $0°$. Given that our system is aimed at working with surrounding ambient light (noise), we borrow the design from [3], which uses a lens at the receiver to reduce the FoV, and thus, limit the noise level.

Overall, our analysis uncovers two key design guidelines. First, for the emitter-transmitter link. Direct sunlight, flashlights and light bulbs –in that order– are preferred due to their directionality. Diffuse (cloudy) daylight is the least ideal condition in spite of being the second most powerful source (after direct sunlight). Second, for the transmitter-receiver link. The more directional the light source is, the more critical is to use mirror-like reflectors. The only case where diffuse surfaces are preferred is when the impinging light is diffuse as well.

3 Transmitter Design

3.1 LC limitations

Most passive-VLC systems using either transmissive [3, 30] or reflective (backscattering) principles [13, 29] rely on liquid crystal shutters (LCs) as the modulating surface. Unlike liquid crystal displays (LCDs), LCs do not have embedded light sources. LCs are readily available, economical, and power efficient, but they suffer from two intrinsic limitations.

3.1.1 Limitation 1: High signal attenuation

LCs only allow a single polarization direction to pass through. All other directions are either fully or partially attenuated. Ambient light, however, is not polarized. This implies that only half of the power can pass through a linear polarizer. On the other hand, DMDs have microscopic mirrors with a high reflection coefficient and are polarization insensitive. For example, the DLP2000 module from Texas Instruments has an efficiency of 97% [9]. Thus, considering the same modulating area and incoming illuminance, DMDs radiate almost 100%

more light than LCs, which can be exploited to increase the range or the data rate of passive links.

3.1.2 Limitation 2: Limited bandwidth

The rise and fall times of commercial LCs take a few ms, as shown in Fig. 4d. These times limit the bandwidth to be under 1 kHz. Furthermore, LCs combine two different operation principles, an electrical signal for the rise time and mechanical inertia for the fall time. This asymmetric operation makes the fall time much slower and it is usually the main bottleneck to increase the bandwidth. Active research has been carried out to squeeze as much data rate as possible from that limited bandwidth, but community efforts are still restricted to around 1 kbps for single-cell designs [3, 7, 13, 29] and 8 kbps for more sophisticated multi-cell designs [25, 28]. DMDs, on the other hand, use the same (fast) operating principle for the rise and fall times. We exploit this fast switching speed to increase the data rate of passive links by an order of magnitude or more.

3.2 DMD basics

A DMD is an optical micro-electro-mechanical system (MEMS) that contains between a few hundred thousand and several millions of highly reflective microscopic mirrors of less than 10 microns each. A DMD can be controlled by electrical pulses, which flip each mirror to one of two fixed directions, for example, $+12°$ and $-12°$. DMDs usually come integrated within a sophisticated projector system called Digital Light Processing (DLP). Besides the DMD, the DLP has a lamp, a light absorber and a projection lens, as shown in Fig. 4a. A micro-mirror is *on* if its angle is tilted towards the projection lens, and *off* if the angle is tilted towards the light absorber. All these optical and electrical components are tightly synchronized by the DLP controller.

There are multiple types of DLPs, as shown in Table 4. All these DLPs tackle *Limitation-1* because DMDs have a high reflective coefficient by design, but exploiting the DMDs' potential for higher bandwidth is harder to attain (*Limitation-2*). On one hand, there are inexpensive units, such as the DLP2000 ($\sim €100$), but their screen refresh rate is too slow. The refresh rate can be seen as the equivalent of the rise (or fall) time in an LC. At 120 Hz, the DLP2000 is even

(a) States: ON/OFF (right/left pixel). Reprinted from [12].

(b) Maximum data rate of out-of-the-box DMD

(c) DMD rise/fall time with custom controller
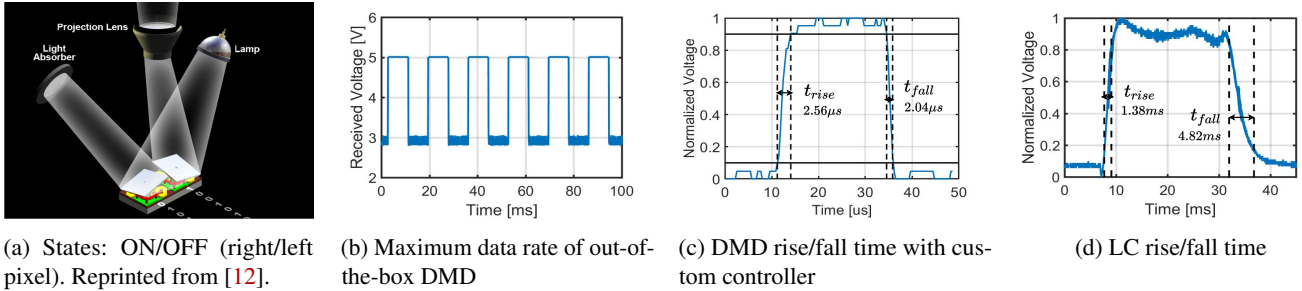
(d) LC rise/fall time

Figure 4: DMD Pixel states and DMD and LC timing characteristics.

slower than the LC shown in Fig. 4d, which provides 320 Hz ($\frac{1000 \times 2}{1.38 + 4.82}$). On the other hand, there are units providing refresh rates above 20 kHz, but with prices beyond €4K, they are prohibitively expensive compared to LC-based systems, which cost a few tens of Euros. A single DMD device (instead of an integrated unit) has comparable cost (€26) to a LC.

The inability to exploit DMDs is an important barrier in passive-VLC. While there are multiple studies utilizing LCs, there are only a few utilizing DMDs. One of those studies uses the same DMD we use, the DLP2000, but attains only a few bits per second because they only use the default refresh rate (120 Hz) and utilize a smartphone camera as a receiver, which is inherently slow [2]. The other studies utilize the more sophisticated DLP4500 (€1100) [10, 11], which provides a maximum refresh rate of 4.2 kHz. Those studies, however, do not exploit that refresh rate for digital communication, but to generate analog signals of just a few tens of Hz (sine, square, triangle, saw-tooth) for localization and audio transmissions. We design a controller for the inexpensive DMD inside the DLP2000 (€26) and increase its refresh rate to 220 kHz, almost a factor of ten faster than the most expensive DLP (DLP9500, €4400). Next, we describe the main limitation of the DLP2000 for ambient light communication, and subsequently, the design of the PhotoLink controller.

3.3 Limitations of inexpensive DMDs.

The DMD from the DLP2000 is the most readily available and economical product, but it is designed for display applications. Hence, for ambient light communication, logical 1s and 0s can only be conveyed as a series of white and black images in a video, which leads to the slow update rate shown in Fig. 4b. In a video application, the pixel's color is obtained by (i) multiplexing RGB beams and (ii) changing the duty cycle of the mirror for each color beam. DMDs provide incredible images, with up to 16.7 million colors, thanks to the fine-grained duty cycle provided by the micro-mirrors. *The micro-mirrors can be flipped at very high speeds between their on/off status*, enabling short operational periods τ, with $\tau \ll T$. These short periods allow a large number of (primary color) combinations. The operation of DMDs, however, is designed for the human eye, which has a slow response. As long as the $3T$ period takes less than 8.3 ms (120 Hz), people will only see high quality videos. Photodiodes, on the other hand, have MHz bandwidth and do not need to capture colors. For
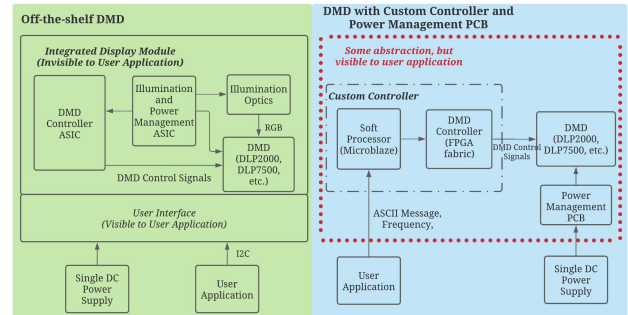

Figure 5: The block diagrams of an off-the-shelf DMD (green, left), and our custom PhotoLink controller (blue, right)

PhotoLink, we need control of τ, not T. Thus, our goal is to remove the controller in the original system and design a new one that gets us as close as possible to the bare fast switching speed of the micro-mirror.

3.4 PhotoLink controller

There are two main obstacles preventing the use of inexpensive DMDs for ambient light communication: no suitable *hardware abstractions* or *operational modes*. Next, we describe each obstacle and the solutions we provide.

3.4.1 Hardware abstraction

Most commercial DLPs do not expose control and power signals to user applications, as illustrated in Fig. 5. There are two ASIC components preventing direct access to these signals: the controller and power management. The *controller* implements the logic to set each micro-mirror and an I2C interface. The interface is the only means to communicate with user applications and hides all control signals. It is therefore impossible to extend beyond the supported frame rate by the controller (120 Hz for the DLP2000). The *power management* controls the DMD power and the integrated RGB light source, which is not needed for ambient light communication.

To increase the refresh rate of the DMD, we remove all hardware components from the original DLP design and use only the DMD. As shown in Fig. 5 (blue side), our main components are: (i) the power management unit, which provides the necessary voltage supplies for different DMD operations without requiring a light source; (ii) an FPGA, which supplies

Table 4: Commercial DMD Module

Name	Clock Rate	Data Bus	Screen Refresh Rate	# of Pixels	Module (DLP) Price	DMD Price	# of pins
DLP2000	60-80 MHz	12(bits)x1	120 Hz	640x360	€109.01	€26.14	42
DLP4500	80-120 MHz	24(bits)x1	4.2 KHz	912x1140	€1106.49	€144.69	80
DLP7000	200-400 MHz	16(bits)x2	32.5 kHz	1024x768	€4144.09	€866.96	203
DLP9500	200-400 MHz	16(bits)x4	23.1 kHz	1920x1080	€4403.30	€2693.38	355

the data and logic for updating the DMD; and (iii) the Microblaze module (soft-processor), which runs on the FPGA and provides a user interface but without hiding the control logic. This interface is used to configure the packet format and the transmitting frequency (explained in section 4).

3.4.2 Operational modes.

Creating a new hardware abstraction is necessary but insufficient to use the DLP2000 for ambient light communication. The next step is to apply the appropriate operational mode to switch the mirrors as fast as possible. The manufacturer does not disclose all the required information to tackle this step, so we base our design on two references: the data sheet of the DMD [9] and a basic description of micro-mirrors [12].

The switching of the mirrors involves two steps: the *memory state* and *micro-mirror state*. In the memory state, the value of each mirror is set (on/off), but the mirror does not tilt. In the micro-mirror state, an actuation pulse tilts the mirrors to their new value. These states define two operational modes.

Individual pixel mode. In this mode, every pixel acts as an individual binary reflector. This allows the DMD to be configured as a fine-grained video projector. The DLP2000 has more than 230 thousand pixels, whose memory has to be written sequentially. As a result, the memory state takes a few hundred μs before any actuation (transmission) can be performed.

Global mode. Considering that the bulk of the delay is in the *memory state*, it would be ideal to by-pass it. In ambient light communication, a fine-grained control of the DMD is not necessary, as photodiodes are used as receivers[2] It is sufficient to update all pixels at once and use the DMD as a single-pixel device, which we dub the global mode. In this mode, we do not write the memory of each pixel, but instead write a global '0' or '1' to all pixels. Attaining this operation requires a careful coordination of various signals[3], but the bandwidth increases dramatically compared to the original DLP design, as shown in Fig. 4c: 60 Hz vs 217.4 kHz, a factor of 3600+[4]. Compared to the LC, the global mode reduces the rise time by a factor of 540 (2.56 μs vs. 1.38 ms) and the fall time by a factor of 2360 (2.04 μs vs. 4.82 ms), which translates to almost a 1350 increase in bandwidth.

3.4.3 Summary of contributions.

Our novel controller allows inexpensive DMDs to be decoupled from their integrated video-projection system. We design a global mode to take full advantage of the fast switching times of micro-mirrors. Compared to LCs, our approach increases the transmitter bandwidth by more than three orders of magnitude. Our controller also achieves a higher refresh rate, even when compared to the high-end DLPs shown in Table 4. Since all DMDs manufactured by TI follow the same operating principles [12], our controller's design would also apply to those DLPs, which could allow them to increase their refresh rates to attain even a better performance than the one obtained with the low-end DLP2000.

4 Optical Link

4.1 Modulation

The majority of modulation schemes fall within two categories: amplitude-based [13, 29] and frequency-based [3]. Amplitude-based methods work well in dark scenarios but are prone to errors when external light sources are present. Frequency-based modulation, on the other hand, has the inherent property of being more resilient to external noise. However, prior LC studies using frequency-based methods had difficulties creating stable periodic signals because the rise and fall times of LCs are asymmetric [3]. DMDs have symmetric times, which allows the generation of stable periodic signals.

To increase the data rate, we use M-ary FSK (MFSK) with two bits per symbol. This high frequency band of PhotoLink (217.4 kHz) allows us to define different modulation parameters and data rates, as shown in Table 6. For example, for a data rate of 30 kbps, we set the four modulating frequencies to 15 kHz, 30 kHz, 45 kHz and 60 kHz. The different modulation parameters permit a thorough evaluation of PhotoLink under different ranges and with different receivers, as discussed in the next section. To avoid abrupt transitions between two frequency signals, the transition between the MFSK frequencies only occurs after a full oscillation period, as depicted in Fig. 6. Considering that the only prior work using MFSK for passive-VLC is [3], we use it as a baseline for comparison. We implement a similar data link layer (shown in Table 5) and receiver design (shown in Fig. 7b and described in Sec. 5). Our packet starts with a SYN symbol (00010101) that uses only the lower transmitting frequencies (00 & 01). These low frequencies have the highest amplitude, and hence, it is easier for the receiver to discover the signal and synchronize to the phase of the transmitter. The ASCII payload is preceded by a STX (Start of Text, 00000010) and followed by ETX (End

---

[2]To take advantage of the individual pixel model, a camera has to be used as a receiver, which is slow (hundreds of frames per second) and requires the use of large screens as transmitters to be efficient.

[3]The hardware and firmware of our controller will be made open source.

[4]The refresh rate of the DLP2000 is 120 Hz, which considers only the time taken by the rise or fall time, the bandwidth considers both times.

Table 5: The structure of the data link layer.

00010101	00000010	ASCII Byte Array	00000011	00010111	00010101
SYN	STX	ASCII Text Message	ETX	ETB	SYN

Table 6: Parameters for different bit rates.

Bit rate	Symbol	Frequency	# of cycle
24 kbps/30 kbps/ 40 kbps/60 kbps/ 80 kbps/100 kbps	00	12/15/20/30/40/50 kHz	1
	01	24/30/40/60/80/100 kHz	2
	10	36/45/60/90/120/150 kHz	3
	11	48/60/80/120/160/200 kHz	4
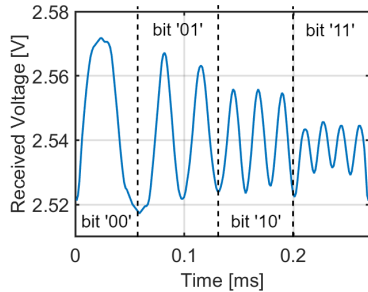
Figure 6: The received signal for different symbols for 100 kbps. Each symbol carries two bits.

of Text, 00000011) and ETB (End of Transmission Block, 00010111).

4.2 Demodulation

The receiver knows the transmitting frequencies and takes the following steps to demodulate the signal.

Preamble detection: A sliding window, equivalent to one symbol, applies a Fourier transform (FFT) to the received signal and decodes the symbol. Every time a byte (four symbols) is decoded, the byte is compared to SYN.

Data demodulation: After a SYN byte is identified, the receiver decodes the incoming message using the same FFT process. If an ETX is received, the packet transmission ends.

Phase correction: If a received two-bit symbol is '00', during the preamble detection or data demodulation, the receiver leverages the presence of this high-amplitude symbol to synchronize to the phase of the transmitter and adjust to any frequency shift that could have been induced by the channel.

5 Evaluation

Our transmitter runs the methods described in Sec. 3 using a custom FPGA controller board and a custom PCB with power management circuits for the DMD. Next, we evaluate our simulation toolbox and controller under various aspects.

5.1 Receiver Design & Data Rate

The design of a low-power optical receiver needs to balance a trade-off between gain and bandwidth. If we optimize for sensitivity (high-gain), small changes in light intensity can be detected, which is advantageous for long-distance communication; but the response is slow (low-bandwidth), which limits the ability to decode high frequency carriers. The opposite trade-off holds for a high-bandwidth receiver. A low-

(a) Transmitter setup (b) Receiver setup
Figure 7: Transmitter and receiver setup
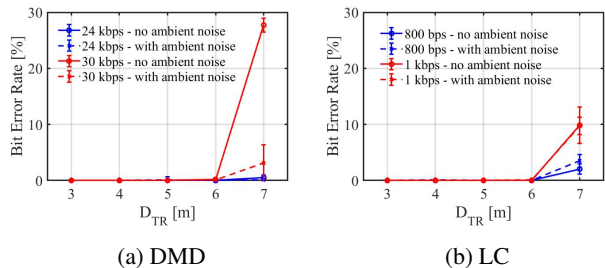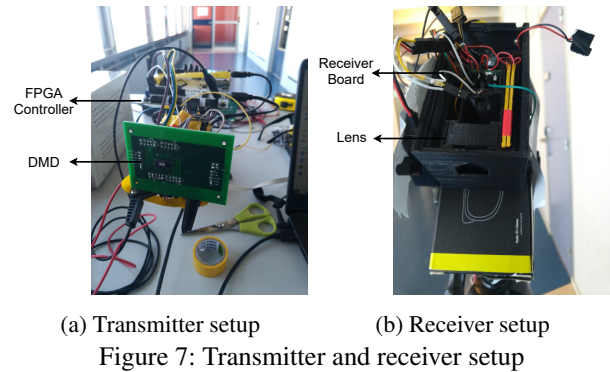

(a) DMD (b) LC
Figure 8: Bit error rate of DMD and LC with artificial light

bandwidth receiver is not a concern when LCs are used as transmitters, as the bandwidth of the LC is low, but it can severely restrict the performance of DMDs. In this subsection, we compare the performance of PhotoLink with LCs used in state-of-the-art studies. we quantify the performance of PhotoLink with two receivers, one optimized for LC operation and the other for DMD. LCs can be used in different ways depending on the type of application: as part of a reflective tag, where either a diffusive or retro-reflective material is placed behind the LC to reflect light, or as part of a transmissive tag, where the LC is used solely as an optical shutter without additional reflective surfaces. To ensure a fair comparison between DMDs and LCs, in the following evaluation, we carry out experiments with both optical devices, but without adding any additional surfaces.

Experiment 1: Receiver optimized for LCs. PhotoLink increases the data rate by a factor of 30.

In this experiment, we use a receiver similar to the one used in [3], consisting of a convex lens with a diameter of 2.5 cm and a TEPT4400 photosensor placed at the focal distance of the lens. The TEPT4400 is a high-gain low-bandwidth photoresistor well-suited for long-range communication with LCs. Using the same illumination environment, we test this receiver using a DMD and an LC as transmitters. The LC and the DMD have the same physical setup, surface area, modulation and demodulation schemes.

Table 7: Rise and fall time for different sensors and resistors

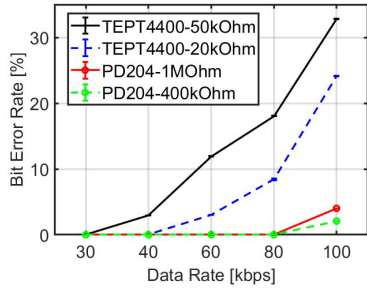| Photoreceiver	Feedback resistor	rise time	fall time
TEPT4400	69/50/20 kΩ	100/64/24 μs	140/105/48 μs
PD204-6C	1000/400/100 kΩ	6.4/3.5/2.5 μs	6.2/3.2/2.1 μs

Figure 9: Bit error rate with different sensors and resistors.

We evaluate the DMD and LC in two scenarios. First, we use a bike flashlight (Simson USB Headlight "Future") to illuminate the transmitting surface in a dark room, such that repeatable experimental results can be obtained. The flashlight is placed 1 m away from the transmitter, and the illuminance at the transmitter is 1800 lx. Then, we use the same setup but turn on the indoor lights located on the ceiling and allow natural ambient light to enter the room (in addition to the flashlight). These additional light rays are not aligned with the receiver and act as ambient noise. The illuminance of the ambient noise (excluding the flashlight) is around 700 lx at the transmitter.

In each experiment, a "Hello world!" packet is sent 100 times. Each experiment is repeated 30 times, and the mean and standard variation of the bit error rates are shown in Fig. 8. With the DMD, we obtain an average BER of less than 1% at 7 m for a data rate of 24 kbps and 6 m for a data rate of 30 kbps. With the LC, we obtain an average BER of less than 1% at 6 m for a data rate of 800 bps and 1 kbps. The data rates achieved with the LC are in line with what has been reported for single-pixel systems [3, 7, 13, 29]. Overall, under the same illumination and modulation conditions, DMDs achieve a data rate of more than 30 times that of LCs.

Experiment 2: Receiver optimized for DMDs. PhotoLink increases the data rate by a factor of 80.

Considering that the maximum data rate achieved by the SoA is 8 kbps [28], a 30 kbps link is a significant improvement. However, using a receiver optimized for LCs does not exploit fully the capabilities of DMDs. Note from Table 6 that the maximum frequency used for a 30 kbps data rate is 60 kHz, but as stated in section 3, the global mode can reach frequencies above 200 kHz. The limitation of low-bandwidth sensors is that they cannot capture fast transitions: even though the *transmitted* signal has rise and fall times below 3 μs (Fig.1c), the *received* signal delivers rise and fall times above 100 μs.

At the core of this phenomenon are two parameters, the parasitic capacitance C_P, which is inherent to the sensor and cannot be modified; and the feedback resistor R_F, which can be modified. A large R_F and C_P improve the receiver's SNR (high-gain), but reduces the bandwidth. We analyze the effect of the feedback resistor on two photosensors: the TEPT4400 (high C_P, low-bandwidth) and the PD204-6C (low C_P, high-bandwidth). Table 7 shows the ability of each receiver-resistor pair to measure the fast DMD transitions for the rise and fall times. The first pair is the configuration used for LCs in [3], and thus, we use it as our baseline. We can see that the PD204 has a bandwidth that is big enough to capture transitions in the few microseconds range.

To showcase the importance of designing an optimal receiver for DMDs, we select four pairs from Table 7, and repeat the same experiment and setup described in Sec. 5.1 but for a fixed distance d_{tr}=2 m. The results are presented in Fig. 9. We know from *Experiment 1* that the TEPT4400 with a 69 kΩ resistor can attain 30 kbps (baseline). A 50 kΩ resistor is not low enough to increase the bandwidth significantly, but a resistor of 20 kΩ can increase the data rate to 40 kbps. The lower capacitance of the PD204, however, increases the bandwidth to a value that is high enough to double the data rate, reaching 80 kbps. Note that the improvement in data rate comes at the cost of reducing the range (lower gain). For the TEPT4400, the range is reduced from 6 m (30 kbps) to 2 m (40 kbps). For the PD204 with 100 kΩ (last pair in Table 7), the data rate reaches 100 kbps but at ranges shorter than 2 m, and thus, is not presented in Fig. 9. The limited range, however, is not a fundamental problem because it can be increased by adding more amplifier stages at the receiver (our receiver has a single stage) or by using focusing lenses at the transmitter. In the case of the LCs, the bandwidth of any photodetector is much higher than the the bandwidth of the LCs, however, that is not the case for DMDs. We expect that an even higher data rate can be achieved if a photodetector with a high gain bandwidth is used together with multiple stages of signal amplification. The data rate, on the other hand, has been a fundamental limitation for passive-VLC and PhotoLink provides a ten-fold improvement over the most sophisticated LC-system in the SoA. Regarding the cost, DMD-based and LC-based systems can make use of the same photo-receivers, a DMD (€ 28.62) costs € 22 more than an LC (€ 6.56). We use an Artix-7 FPGA, which cost € 50, but a less expensive controller can be used as well. A microcontroller that costs € 13.6 was used in Luxlink [22].

5.2 Analyzing the Luminous Flux

Our work has two main contributions, the controller evaluated in the prior subsection and the toolbox presented in section 2. The main insight of our toolbox is the importance of maintaining the luminous flux throughout the optical link. To capture this phenomena, we consider two scenarios.

Scenario 1: Normalized flux (indoor setup). In this scenario, we use the baseline receiver (TEPT4400 with a 69 kΩ resistor) and transmit a fixed carrier frequency. The frequency is empirically chosen to be 30 kHz because this signal can be clearly detected at 4 m without saturating the receiver at 1 m. To calculate the amount of luminous flux maintained

in the optical link, the signal intensity measured at 4 m is normalized with respect to the intensity measured at 1 m for *the same* light source. This normalization process and careful setup quantify the impact of the radiation pattern of each light source independent of its emitted power.

Under this setup, we evaluate four different light sources *indoors*, as shown in Table 8, and simulate the same illumination conditions with our toolbox. In the test setups, the direct and diffuse sunlight arrive at the DMD through a large glass window. To obtain realistic simulation results, we apply the parameters in Table 9, which correspond to the actual physical properties of PhotoLink. The normalization method is also applied to the simulations[5], and the results are shown in Fig. 10. The plots show a strong agreement between the experimental and simulated flux under all illumination conditions. With diffuse sunlight, we are not able to detect a signal even at 1 m, despite measuring a 2000 lx illuminance on the surface of the DMD. This aligns with our analysis in Sec. 2, which states that diffuse light has the lowest performance with reflective surfaces. The results also show that direct sunlight performs best at retaining the luminous flux (losing 30% at 4 m), followed by artificial lights (losing more than 80%). And with artificial lights, more luminous flux is retained at the receiver when the light is placed further away from the transmitter (setup 2). All these results are in agreement with the insights provided by our model in Sec. 2.

Scenario 2: Absolute flux (outdoor setup). In this scenario, we do not perform a normalization process, instead, we transmit 100 packets of "Hello world!" at 30 kbps and present the received voltage and BER. We evaluate the two best light sources identified by our toolbox, flashlight and direct sunlight. The evaluation with direct sunlight is done *outdoors* during a clear day with good sunlight (several thousand lux), and then, moving the setup indoors and placing a flashlight in a dark room.

With sunlight, a BER of $0.9x10^{-3}$ is achieved at 1 m and a BER of $0.8x10^{-3}$ is achieved at 2 m. The errors can be attributed to the fact that a link in the outdoor environment is subjected to occasional disturbance. With flashlight, the BER is 0 at 1 m, however, at 4 m, the BER increases to $19.4x10^{-3}$. In Fig. 11, we present a direct comparison of the flux reaching the receiver with the flashlight and sunlight using the SYN symbols in the packet. At 1 m, the flux reaching the sensor with the flashlight is slightly lower than that with sunlight (around 0.18 V vs. 0.2 V), which shows that the flashlight and sunlight result in similar voltage range. However, at 4 m, the luminous flux reduces by 60% with flashlight, due to a less directional pattern, while direct sunlight loses only 10%[6]. This result highlights the importance of expanding passive-

---

⁵Since photodiodes have a quasi-linear response to light intensity, we assume a linear correlation between the obtained signal strength in the toolbox and the voltage obtained in our experiment.

⁶Note that the flashlight loss is higher than the loss predicted in Fig. 10 for 1 m because we place the light closer to the DMD, 30 cm instead of 1 m

Table 8: Measuring the power drop-off with respect to distance

Setup	Light Source	D_{LM}	D_{TR}	Measured Normalized Signal	Simulated Normalized Signal
1	Direct Sunlight	N/A	1 m and 4 m	0.70	0.73
2	Flashlight	4 m		0.17	0.20
3	Flashlight	1 m		0.04	0.06
4	Diffuse Sunlight	N/A		N/A	N/A

Table 9: Key parameters used in simulator

Light Source	Dimension	2.7 m x 2.7 m
	Half Angle	30°
Modulating Surface	Dimension	4.8 mm x 2.7 mm
	Light-absorbing area	8 mm by 8 mm
	Spreading Angle	0.3°
Receiver	Lens Dimension	2.5 cm
	Tangent Sphere Radius	(4 cm, -5 cm)
	Lens Material	bk7
	FoV	1°
	Photodiode Diameter	3 mm

VLC studies towards the exploitation of natural light.

5.3 Issues with DMDs

DMDs have not been designed for ambient light communication, and hence, present some limitations. We now discuss what we consider the main shortcomings of this MEMS technology for passive-vlc.

Issue 1: Directionality. DMDs operate with two fixed angles, which raises up the issue of directionality. If the light source changes its location, the impinging light rays will no longer be aligned with the predefined angles at the DMD, breaking the link. This issue can be overcome with light concentrators. As a proof of concept, we build a simple concentrator with two optical components, as show in Fig. 12a. The first component is a Compound Parabolic Concentrator (CPC). The CPC is a special parabolic lens that collects light from different angles and concentrates it on a small output area. We use a CPC with an input and output circular area of 14 mm and 4 mm diameter respectively, and a concentration factor of 10. The second component is a ball lens of 8 mm diameter, which is used as a coupler and collimator, to further focus the collected light. We manufacture a 3D-case to align the CPC and the ball lens, and aim its output to the DMD.

Fig. 12b presents the results obtained with and without the light concentrator. A flashlight is positioned at different incidence angles and the signal strength is measured at the receiver. Without the concentrator, the signal strength decays below 0.9 with deviations around +/- 1 degree. With the concentrator, the signal remains above 0.9 for angles around +/- 10 degrees. This is a simple implementation, more sophisticated designs can increase the FoV to any desired degree. Thus, while an ideal DMD design for passive-VLC could consider flexible angles, it is not a strict requirement.

Issue 2: Power consumption. Perhaps the most limiting factor of current DMD designs is the relationship between its small area and relatively high power consumption. The
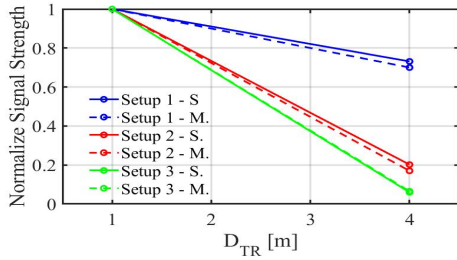
Figure 10: Simulated and measured voltage dropoff.
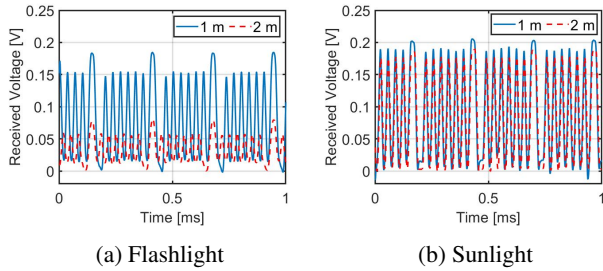

(a) Flashlight   (b) Sunlight

Figure 11: Signal strength

performance of passive-VLC depends on the area of the transmitting surface. For example, a standard light bulb consumes 1 watt to emit 90 lumen, But with an area of 13 mm^2, the DLP2000 would emit only between 0.1 and 0.5 lumen.

Regarding the power consumption, current DMD designs have two levels of overhead. The first level is related to the *memory state*, which is not required for PhotoLink and consumes 57 mW in the DLP2000. The second level is the actuation of the micromirrors and consumes 45.5 mW. We are, thus, left with a surface that emits between 0.1 and 0.5 lumen (depending on daylight conditions), while consuming 45 mW. Considering that LCs consume less than 1 mW, and that low-power LEDs consume less than 100 mW, it is central to consider power consumption in the comparison with LCs and LEDs. To perform that analysis, let us consider a low-power LED that has been used in prior VLC studies [8], the VLMB1500, which consumes 75 mW and emits 0.2 lumen. We perform a theoretical comparison between LCs, DMDs and LEDs based on the Shannon-Hartley theorem $C = B \log_2(1 + SNR)$. The analysis assumes a luminous flux of 0.2 lumen for the DMD.

First, let us consider LCs, which have areas that are two orders of magnitude bigger than DMDs, and hence, receive two orders of magnitude more lumen. Assuming that the LC receives 20.2 lumen on its incoming area (20 from the light source and 0.2 from the 'extra' LED), the outgoing flux would be 10.1 lumen because LCs cut the power in half (Limitation-1 in section 3). Hence, the SNR of an LC-system would increase by a factor of 50 (10.1/0.2), but the SNR only contributes logarithmically to the capacity. Overall, the extra SNR would contribute with a factor of 6, compared to the factor of 1350 contributed by the BW of the DMD, making the DMD still two orders of magnitude more competitive.
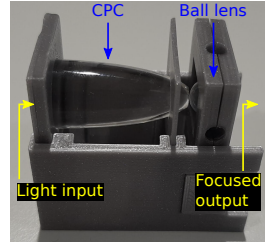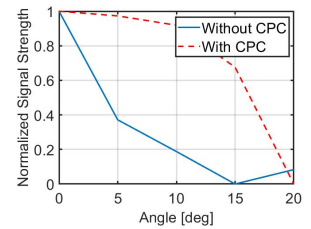
(a) Design   (b) Evaluation

Figure 12: CPC

To consider the option of using the LED with active-VLC, we measure its rise and fall times, which are 3.5 µs and 1.6 µs: a period of 5.1 µs compared to the period of 4.6 µs for DMDs. Recalling that the LED and DMD emit a similar lumen, the DMD is only slightly more competitive. This result, however, should consider that current DMDs are not designed for ambient light communication. The mirrors of the DLP have a size of microns and use *electrostatic* actuation, larger area mirrors (bigger than 2 mm) "benefit from *electromagnetic* actuation proportional to the mirror area" [17], leading to bigger surfaces with lower power consumption.

Thus far, the passive-VLC community has faced a major obstacle, even with big LC surfaces, no system can provide data rates above 10 kbps. PhotoLink shows that current (suboptimal) DMDs can provide 100 kbps. Synergies with MEMS researchers could enable the design of bigger modulating surfaces to create wireless networks operating solely with natural light and with low power budgets.

6 Related Work

Passive VLC systems using LCs. There have been several studies on passive VLC systems, which are summarized in Table. 10. To date, LCs have been widely used as an optical transmitter in SoA passive VLC systems. There are two categories: one uses LCs in combination with retro-reflectors, where the light source and the receiver are co-located; the other adopts only the LC as a transmitter, where the light source and the receiver can be placed at different locations, opening up the possibility to take advantage of natural light.

The studies in the former category typically have a constrained data rate and range. The earlier studies [13, 29], achieve a data rate of 0.5 kbps and 1 kbps with ranges up to a few meters. More recently, RetroI2V [25] achieves a range of 80 m. However, it uses a powerful 30 W light to achieve a data rate around 1 kbps. RetroTurbo [28] has a surface area of 66 cm^2 and uses an advanced modulation scheme to overcome the slow time response of LCs. RetroTurbo achieves a data rate of 8 kbps with a moderate light source (4W flashlight) up to 7.5 m [25]. However, retro-reflectors cannot be used with ambient light, as the light source and the receiver have to be co-located. On the other hand, the studies in the latter category take advantage of the strong illumination from sunlight and are able to achieve a long range without a dedicated illuminator, such as in the case of Luxlink [3] and Chromalux [7]. Luxlink is able to reach a range of 65 m with sunlight, but the

Table 10: Comparison of PhotoLink with the most relevant systems in the state of the art.

Name	O_L	O_L Power or Illuminance	D_{LT}	O_T	Surface Type	O_R	FoV	Data Rate	Range
RetroVLC [13]	LED	12 W	Variable[1]	LC+RR[2]	Specular	Photodiode	50°	0.5 kbps[2]	2.4 m
PassiveVLC [29]	Flashlight	3 W	Variable	LC+RR	Specular	PD	4°	1 kbps	1 m
RetroTurbo [28]	Flashlight	4 W	Variable	LC+RR	Specular	PD	20°	8(4) kbps	7.5(10.5) m
RetroI2V [25]	Flashlight	30 W	Variable	LC+RR	Specular	PD	30°	125(1000) bps	101(80) m
Chromalux [7]	Sunlight(Direct) / Flashlight	3-6 klx / 400-700 lx	N/A / N/S	LC and Metal Sheet	Specular	Color Sensor	Variable	1 kbps	50 m / 10 m
Luxlink [3]	Sunlight(Direct) / LED	10-26 klx / 2 klx	N/A / N/S[3]	LC and Diffuser	Diffuse	PD	1°	80 bps / 1 kbps	65 m / 3 m
TwSL [4]	Sunlight(Diffuse)	3 klx	N/A	Paper	Diffuse	PD	N/S	127 bps	4 m
[10]	LED	15 W	c̃ms	DMD	Specular	PD	N/S	4.2 kbps	170 cm
[11]	LED	15 W	c̃ms	DMD	Specular	PD	N/S	9 bps	2.5 m
[2]	Sunlight(Direct)	330 lux	N/A	DMD	Specular	Camera	N/S	1 bps	60 cm
PhotoLink	Flashlight	1800 lx	1 m	DMD	Specular	PD	1°	30(80) kbps	6(2) m

[1] For work involving retro-reflectors as a transmitter, $D_{LT} = D_{TR}$.
[2] RR stands for Retro-reflectors.
[3] For work involving retro-reflectors, uplink data rate is quoted.
[4] N/S stands for 'not specified'.

data rate is limited to 80 bps. It also demonstrates that with an LED, a data rate of 1 kbps can be achieved up to 3 m. Chromalux [7] takes advantage of the transient state in LCs, and is able to achieve a range of 50 m with a data rate of 1 kbps with sunlight, and up to 10 m with a flashlight. While LCs are energy efficient, they suffer from a high attenuation loss due to the use of polarizers, and a limited bandwidth because of its slow rise and fall times. On the other hand, a higher data rate can be achieved with DMDs, but using more power. And in addition to demonstrating a novel system, we provide an analytical framework to understand the performance of different studies.

Applications of DMDs. Like LCs, the main application of DMDs is video projection, and thanks to their competitive advantages (high reflective efficiency and switching times) they dominate the market. But DMDs are also used in other applications: microscopy, holography, data storage, and also as spatial modulators with lasers [6]. The use of DMDs for ambient light modulation, however, is restricted to a handful of studies involving localization [10] and communication [2,11]. And all these studies suffer from a limited data rate (1 bps, 9 bps and 4 kbps), as well as a limited communication range (60 cm, 2.5 m and 170 cm). These systems use the off-the-shelf DMD controllers with their default refresh rates, which fail to take advantage of the fast switching times of the DMDs.

Channel modelling for VLC systems. There have been an array of studies on channel modelling techniques for indoor active VLC systems [21], several of which are ray-tracing based [15, 16]. The focus of those studies is to achieve an accurate impulse response considering the dynamics of the VLC system and its indoor environment. They remain a theoretical exercise in most cases, as an accurate description of the indoor space is difficult to obtain. This differs from our work,

as our study focuses on the interactions between different types of surfaces and ambient light.

Ambient RF backscatter systems. In RF backscatter, passive devices can communicate with each other utilizing surrounding RF sources. The first study exploited TV tower signals and showed a data rate of 1 kbps at distances of 2.5 feet outdoors and 1.5 feet indoors [14]. Subsequent studies have shown that WiFi, BLE and LoRa signals can also be backscattered, attaining even higher data rates and/or ranges [1, 24]. RF backscattering is an exciting area but requires *man-made* signal (radio towers and antennas), and antennas typically have a limited bandwidth. Ambient light backscattering not only allows exploiting a different part of the electromagnetic spectrum but it can also exploit *natural* sunlight.

7 Conclusion

In this work, we propose an optical model to analyze ambient light communication, and based on the insights it provides, we explore the use of a DMD as an optical transmitter. We propose a novel platform that optimizes the retention of the luminous flux to attain the best optical performance. This approach allows us to achieve a data rate that is 30 times higher compared to LCs under the same working conditions. Furthermore, with optimally designed receivers, the data rate reaches 80 kbps. While current DMD designs still face limitations to operate with ambient light, it is a component that expands the possibilities of the nascent area of Passive-VLC.

Acknowledgments

References

[1] Dinesh Bharadia, Kiran Raj Joshi, Manikanta Kotaru, and Sachin Katti. Backfi: High throughput wifi backscatter. volume 45, page 283–296, New York, NY, USA, aug 2015. Association for Computing Machinery.

[2] Roy Blokker. Communication with ambient light using digital micromirror devices. Master's thesis, Delft University of Technology, 2021.

[3] Rens Bloom, Marco Zúñiga Zamalloa, and Chaitra Pai. Luxlink: Creating a wireless link from ambient light. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*, SenSys '19, page 166–178, New York, NY, USA, 2019. Association for Computing Machinery.

[4] Rens Bloom, Marco Zuniga, Qing Wang, and Domenico Giustiniano. Tweeting with sunlight: Encoding data on mobile objects. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pages 1324–1332, 2019.

[5] Cisco. Cisco annual internet report - cisco annual internet report (2018–2023) white paper.

[6] Dana Dudley, Walter Duncan, and John Slaughter. Emerging digital micromirror device (dmd) applications.

[7] Seyed Keyarash Ghiasi, Marco A. Zúñiga Zamalloa, and Koen Langendoen. A principled design for passive light communication. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, MobiCom '21, page 121–133, New York, NY, USA, 2021. Association for Computing Machinery.

[8] Tilahun Zerihun Gutema, Harald Haas, and Wasiu O. Popoola. Bias point optimisation in lifi for capacity enhancement. *Journal of Lightwave Technology*, 39(15):5021–5027, 2021.

[9] Texas Instrument. Dlp2000 (.2 nhd) dmd datasheet, 2019. https://www.ti.com/lit/ds/symlink/dlp2000.pdf.

[10] Motoi Kodama and Shinichiro Haruyama. Visible light communication using two different polarized dmd projectors for seamless location services. In *Proceedings of the Fifth International Conference on Network, Communication and Computing*, ICNCC '16, page 272–276, New York, NY, USA, 2016. Association for Computing Machinery.

[11] Motoi Kodama and Shinichiro Haruyama. Pulse width modulated visible light communication using digital micro-mirror device projector for voice information guidance system. In *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 0793–0799, 2019.

[12] Benjamin Lee. Introduction to ±12 degree orthogonal digital micromirror devices (dmds), 2018. https://www.ti.com/lit/an/dlpa008b/dlpa008b.pdf.

[13] Jiangtao Li, Angli Liu, Guobin Shen, Liqun Li, Chao Sun, and Feng Zhao. *Retro-VLC*: Enabling battery-free duplex visible light communication for mobile and iot applications. In Justin Manweiler and Romit Roy Choudhury, editors, *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications, HotMobile 2015, Santa Fe, NM, USA, February 12-13, 2015*, pages 21–26. ACM, 2015.

[14] Vincent Liu, Aaron Parks, Vamsi Talla, Shyamnath Gollakota, David Wetherall, and Joshua R. Smith. Ambient backscatter: Wireless communication out of thin air. *SIGCOMM Comput. Commun. Rev.*, 43(4):39–50, aug 2013.

[15] Francisco J. Lopez-Hernandez, Rafael Perez-Jiminez, and Asuncion Santamaria. Ray-tracing algorithms for fast calculation of the channel impulse response on diffuse IR wireless indoor channels. *Optical Engineering*, 39:2775–2780, October 2000.

[16] Farshad Miramirkhani and Murat Uysal. Channel modeling and characterization for visible light communications. *IEEE Photonics Journal*, 7(6):1–16, 2015.

[17] Pamela Rae Patterson, Dooyoung Hah, Makoto Fujino, Wibool Piyawattanametha, and Ming C. Wu. Scanning micromirrors: an overview. In Yoshitada Katagiri, editor, *Optomechatronic Micro/Nano Components, Devices, and Systems*, volume 5604, pages 195 – 207. International Society for Optics and Photonics, SPIE, 2004.

[18] Yury Petrov. Optometrika, howpublished = https://github.com/caiuspetronius/optometrika, 2014.

[19] Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, June 1975.

[20] PureLiFi. http://purelifi.com/, 2021.

[21] A.M. Ramirez-Aguilera, J.M. Luna-Rivera, V. Guerra, J. Rabadan, R. Perez-Jimenez, and F.J. Lopez-Hernandez. A review of indoor channel modeling techniques for visible light communications. In *2018 IEEE 10th Latin-American Conference on Communications (LATINCOM)*, pages 1–6, 2018.

[22] Yu-Xuan Ren, Rong-De Lu, and Lei Gong. Tailoring light with a digital micromirror device. *Annalen der Physik*, 527(7-8):447–470, 2015.

---

[23] Nils Ole Tippenhauer, Domenico Giustiniano, and Stefan Mangold. Toys communicating with leds: Enabling toy cars interaction. In *2012 IEEE Consumer Communications and Networking Conference (CCNC)*, pages 48–49, 2012.

[24] Nguyen Van Huynh, Dinh Thai Hoang, Xiao Lu, Dusit Niyato, Ping Wang, and Dong In Kim. Ambient backscatter communications: A contemporary survey. *IEEE Communications Surveys Tutorials*, 20(4):2889–2922, 2018.

[25] Purui Wang, Lilei Feng, Guojun Chen, Chenren Xu, Yue Wu, Kenuo Xu, Guobin Shen, Kuntai Du, Gang Huang, and Xuanzhe Liu. Renovating road signs for infrastructure-to-vehicle networking: A visible light backscatter communication and networking approach. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, MobiCom '20, New York, NY, USA, 2020. Association for Computing Machinery.

[26] Zixiong Wang, Dobroslav Tsonev, Stefan Videv, and Harald Haas. On the design of a solar-panel receiver for optical wireless communications with simultaneous energy harvesting. *IEEE Journal on Selected Areas in Communications*, 33(8):1612–1623, 2015.

[27] Maury Wright. Philips lighting deploys led-based indoor positioning in carrefou, 2015. https://goo.gl/a0tGIj.

[28] Yue Wu, Purui Wang, Kenuo Xu, Lilei Feng, and Chenren Xu. Turboboosting visible light backscatter communication. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 186–197, New York, NY, USA, 2020. Association for Computing Machinery.

[29] Xieyang Xu, Yang Shen, Junrui Yang, Chenren Xu, Guobin Shen, Guojun Chen, and Yunzhe Ni. Passivevlc: Enabling practical visible light backscatter communication for battery-free iot applications. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, MobiCom '17, page 180–192, New York, NY, USA, 2017. Association for Computing Machinery.

[30] Zhice Yang, Zeyu Wang, Jiansong Zhang, Chenyu Huang, and Qian Zhang. Wearables can afford: Lightweight indoor positioning with visible light. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '15, page 317–330, New York, NY, USA, 2015. Association for Computing Machinery.

Whisper: IoT in the TV White Space Spectrum

Tusher Chakraborty, Heping Shi, Zerina Kapetanovic[†], Bodhi Priyantha, Deepak Vasisht[‡], Binh Vu,
Parag Pandit, Prasad Pillai, Yaswant Chabria, Andrew Nelson, Michael Daum, and Ranveer Chandra
Microsoft, [†]University of Washington, [‡]UIUC

Abstract

The deployment of Internet of Things (IoT) networks has rapidly increased over recent years – to connect homes, cities, farms, and many other industries. Today, these networks rely on connectivity solutions, such as LoRaWAN, operating in the ISM bands. Our experience from deployments in multiple countries has shown that such networks are bottlenecked by range and bandwidth. Therefore, we propose a new connectivity solution operating in TV White Space (TVWS) spectrum, where narrowband devices configured for IoT can opportunistically transmit data, while protecting incumbents from receiving harmful interference. The lower frequency of operation extends the range by a factor of five over ISM bands. In less-densely populated area where larger swaths of such bandwidth are available, TVWS-based IoT networks can support many more devices simultaneously and larger transmission size per device. Our early experimental field work was incorporated into a petition to the US FCC, and further work influenced the subsequent regulations permitting the use of IoT devices in TVWS. We highlight the technical challenges and our solutions involved in deploying IoT devices in the shared spectrum and complying with the FCC rules.

1 Introduction

The growth in IoT is accelerating and expanding across a wide variety of industries. Networking has emerged as a fundamental challenge for IoT. Current solutions like LoRaWAN rely on narrowband (NB) connectivity in the ISM bands, such as US915, EU868, CN779, and so on [23]. However, as IoT networks continue to expand, they run into bottlenecks of these networking solutions. Consider an agriculture scenario, where IoT devices are used to enable precision agriculture techniques on farms in remote areas. These farms can span tens of thousands of acres. LoRaWAN has a communication range of up to 2.5 miles [4]. To connect such vast coverage areas, multiple LoRa gateways need to be deployed and maintained, where deploying a single gateway may cost thousands of US dollars. Besides, setting up proper backhaul connectivity for a gateway is cumbersome in remote areas. It holds for several other scenarios, such as oil and gas fields, power grids, wind farms, and so on. Furthermore, ISM bands have a limited bandwidth, e.g., only 8 MHz in EU868 and 26 MHz is US915 where bandwidth allocated for downlink communication is even smaller (see Section 2). Therefore, it becomes challenging to support IoT applications such as

heatmap-based monitoring and plant stress monitoring using cameras, where comparatively larger volumes of data traffic is required to be transmitted over a low data rate long-range network [7, 17, 18]. In such cases, the combination of longer-range and larger available bandwidth is required.

To bridge the gap, we envision enabling IoT networks over the TV white spaces (TVWS). TV white spaces are the allocated, but unused channels in the VHF and UHF broadcast TV bands that can be leveraged for both high and low bandwidth data transmission. There are several advantages of utilizing TVWS spectrum for a NB IoT deployment over ISM bands. As the TV band spectrum consists of lower frequencies than the 800/900 MHz ISM bands, it facilitates longer-range connectivity which extends to 10s of miles, with non-line of sight (NLOS) operations, and even through some obstructions. Consequently, it opens the door of covering a large-area IoT deployment with one or a minimal number of gateways. It even facilitates higher data rates for distant IoT clients which in turn provides power savings on the IoT device. In addition, in less densely populated areas, there are typically several unused TV channels available for use by TVWS devices. The actual number of available channels vary by location. In the aggregate, these available channels can offer a large bandwidth, and hence support for many more simultaneous communication channels and increased traffic.

However, in the way of realizing this vision, the challenges are twofold – regulatory and technical. While operating in a dynamic spectrum as the unlicensed user, the precondition is to ensure the protection of incumbents from receiving harmful interference and the inability to claim protection from interference. Although this challenge has been addressed in the case of unlicensed broadband devices operating in the TVWS [5], it is non-trivial to extend the same to NB devices due to their limited power budget and distinct regulations for NB operation, such as channel occupancy limit (Section 3). We identify three corresponding challenges below:

- First, in a large-area deployment, the data rate of sparsely deployed clients (Figure 7c) served by a single gateway becomes highly variable as the data rate is inversely proportional to the distance. Single configuration setting of slow data rate (longer range) for all the clients, will result in throughput loss and power overhead. On the other hand, mainstream IoT MAC protocols, mainly designed for ISM bands, cannot make the best utilization of wide TVWS spectrum in serving large traffic even using a gateway with multi-data rate support on a single channel (Section 7.2).

- The second challenge is handling the spatio-temporal dynamism in TVWS channel availability and quality. The dynamism is mainly due to the channel occupancy by nearby licensed users (e.g., TV station, wireless microphone, etc.) and unpredictable unlicensed users (e.g., TVWS broadband network). Moreover, the long spatial separation between the gateway and client devices implies that uplink and downlink may operate on different channels with dissimilar quality. Finally, the power constraint of IoT client devices adds a curb on dynamic spectrum access and management.

- The final challenge is to develop an efficient carrier sensing solution to detect the presence of an interfering RF transmission from incumbents – both licensed and unlicensed – in a dynamic spectrum. With LoRa modulation, the devices can communicate even when the signal level is below the ambient RF noise floor. Hence, the conventional approach of simply measuring RF energy level in a NB channel to detect RF interference does not work.

Over the years, we have worked on devising Whisper, an end-to-end IoT network system over the TVWS spectrum which addresses both regulatory and technical challenges. Our early field work, authorized under an experimental license, led to a proposal on NB TVWS device operations which was the part of a broader 2018 petition for rulemaking to the US Federal Communications Commission (FCC) for expanding its rules for TVWS devices. Later, our work supported the FCC's December 2020 decision to adopt regulations on NB white space devices to operate in the VHF and UHF bands below 602 MHz [10]. In addition, we make the following contributions through this work.

Whisper Protocol: We design a new MAC protocol for a star-topology IoT network operating in the TVWS spectrum. Our Frequency Time Division Multiple Access (FTDMA) based design supports larger traffic along with diversity in the data rate of sparsely deployed clients. It leverages a dynamic binary counting table with the linear Diophantine equation for formalizing and optimally limiting the channel occupancy to protect the incumbents. The protocol further incorporates a smart approach for handling the dynamism in the TVWS spectrum given the power limitation of IoT devices.

Whisper Hardware: We design and develop a NB Whisper radio that operates in the continuous spectrum ranging from 150MHz to 960MHz. The radio uses LoRa modulation at the physical layer. Given that and the above-mentioned challenge in corresponding carrier sensing, we further develop a spectrum sensing module that uses a locally generated signal by a Whisper radio to measure the RF interference from the incumbents in individual NB TVWS channels.

Real-world Deployment: Finally, we make a real-world deployment of Whisper as an end-to-end IoT network system for more than 2.5 months. The deployment covers 17 fields in a 8500 acre farm with single gateway and 20 IoT devices. The sensor data collected via Whisper is used by third-party users in multiple agriculture applications including food tracing,

Data type	Area (acre)	#gateways	#clients	Traffic (bytes/hr)	Prominent issue
Sensor	1700	3	11	3.2k	Range
Sensor	350	2	20	0.9k	Range
Sensor	700	3	9	0.5k	Range
Image	8500	2(Abortive)	20	550k	Bandwidth

Table 1: Setup of multiple real-world IoT deployments where Whisper would befit. These are representative data from our deployments across the globe using ISM band LoRa.

data-driven farming [32], and carbon monitoring.

From our real-world deployment, we find that Whisper facilitates at least 5x range improvement over LoRa operating in the 800/900 MHz ISM band and at least 3x over state-of-the-art modulation techniques proposed for NB operation in TVWS [27, 28]. Furthermore, our simulation shows that using only 3 white TV channels (6 MHz each), Whisper can handle at least 5x traffic compared to ISM band.

2 Motivation from Real-world Experience

The need for deploying IoT devices in TVWS spectrum is motivated by the bottlenecks experienced in our real-world deployments using LoRa operating in the ISM bands. We highlight two application scenarios here.

The first application scenario emanates from one of our earlier projects, FarmBeats, that aims to enable data-driven agriculture [32]. To do so, we deploy sensors across a farm and aggregate data in a star-topology LoRa network operating in 800/900 MHz ISM bands. We have made more than 30 research deployments in farms across the globe (including US, South Asia, Europe, Africa, Asia Pacific, etc.) over a period of 4 years. The top three entries in Table 1 are representative of the setup of these deployments. The major challenge we have experienced in these settings is the relatively short range of the communication link compared to the size of a farm and sparsely deployed IoT clients. The average maximum achieved range is 1.12 miles combining both NLOS and LOS settings. Consequently, we need to deploy multiple gateways to cover a farm, even when we need to support just a small number of sensors spread across the farm. Whereas, TVWS spectrum offers a range of tens of miles, and thus, reduces the number of gateways as we show in Section 7.

In the second application scenario (bottom entry in Table 1), we study the feasibility of monitoring plant stress using a camera in US915 ISM band [24]. Each client sends an image of ∼ 25 kB where the gateway can expect at least 22 images per hour. Here, the foremost problem is sending a large number of confirmed-up frames as the ISM band suffers from the paucity of bandwidth in two levels. First, the dwell time restriction of 0.4 sec enforces sending an image in a large number of small uplink frames. It, in turn, increases the load on the downlink (allocated bandwidth is 4 MHz) with a large number of ACKs. For example, if we consider the median uplink data rate (DR2) supported in US915 band, it takes around 230 uplink frames for an image [3]. On the downlink side, with a

comparable data rate (DR12), a gateway can serve ACKs for maximum 7 images per hour without even considering any frame loss and the inefficiency of existing MAC protocols in handling confirmed-up frames [3, 16]. Multiple existing research work report similar problem [7, 17, 18]. Even one of the largest commercial LoRaWAN service providers, The Things Network, recommends finding an alternative platform in such scenarios [22]. Furthermore, with the aforementioned data rates, the maximum range can be up to a mile. In this scenario, TVWS spectrum offers a larger bandwidth that can easily handle the aforementioned traffic (Section 7.2).

3 Regulating NB Operation in TVWS

Although FCC has adopted regulations on NB operation in TVWS spectrum in 2020, we have been working with FCC on it for more than four years. FCC's Office of Engineering Technology granted us several experimental licenses for operating NB IoT TVWS devices in an agricultural setting. From the beginning, the experimental NB TVWS transmitter and network architecture have been designed with the understanding that the primary users of these frequency bands must be protected from receiving harmful interference. Based on the experience gained over the course of the field tests, we filed a petition for rulemaking at the FCC for expanding TVWS operations that included NB. Next, we describe the key regulations mandated by FCC for NB operation in TVWS spectrum [10].

- Incumbents are protected through a geolocation and database method. The location of the NB is provided through an incorporated geolocation decision, typically GPS [9]. The geolocation information is provided to a white spaces database (WSDB). The WSDB combines information on incumbent users from the FCC Licensing and Management System that is updated daily; information from other users input directly, such as wireless microphones that is updated hourly; and a calculation engine that determines the list of available channels for the TVWS device operating at that location.

- A TV channel in the US is 6 MHz wide. The conducted power and power spectral density limits for broadband TVWS devices are based on 100 kHz. Thus, the channel size limit for a NB device is proposed to be 100 kHz. The proposed channel plan requires NB TVWS devices to operate at least 250 kHz from the edge of a 6 MHz TV channel. It implies that NB devices are permitted to operate within 55 possible 100 kHz NB channels in the center 5.5 MHz of each TV channel.

- FCC limits transmissions by a NB TVWS device on each NB channel to a total of 36 sec per hour. It means that different NB channels may be required for the uplink/downlink data communication and interaction with the WSDB.

- The conducted power and power spectral density limits for NB devices per 100 kHz are the same as for broadband
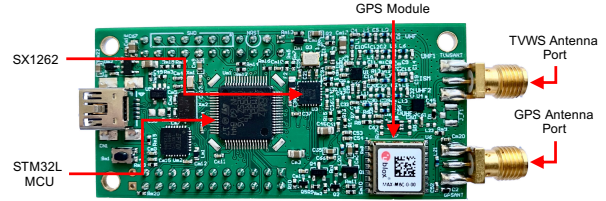

Figure 1: Whisper radio fabricated as an industrial-grade module that can operate from 150MHz to 960MHz.

TVWS devices. Here, the EIRP of a NB TVWS device can be up to 18.6 dBm/100 kHz. The rules for protecting incumbents are based on a scenario where each of the 55 channels of 100 kHz in a TV channel is concurrently being used at its conducted power limit.

We highlight that the NB devices can operate in 174-216 MHz in the VHF band and 470-602 MHz in the UHF band.

4 System Design

Whisper is a new IoT system that can support long-range communication at large-scale in the TVWS spectrum. We design Whisper to have two key components similar to a classic star-topology IoT network: an IoT client radio and gateway. In the following subsections, we describe each component.

4.1 Whisper radio

Whisper requires a radio that can operate over the whole TVWS spectrum in both VHF and UHF bands as mentioned in Section 3. We further intend to use LoRa modulation in the physical layer, which is very popular for long-range and low-power wide area network (LPWAN). Since commercial off-the-shelf LoRa radios are designed to operate over narrow ISM bands, we develop a custom NB IoT radio that can operate over a wider TVWS spectrum. Figure 1 shows the fabricated radio which can operate from 150 MHz to 960 MHz including the upper-VHF, UHF, and ISM bands. For modulation and demodulation of LoRa signal, our radio incorporates an SX1262 radio transceiver manufactured by Semtech [12].

Unlike an off-the-shelf radio designed using SX1262 for narrow ISM bands, our design for a wider TVWS spectrum requires careful consideration of avoiding leakage and harmonics in an adjacent TV channel or other licensed bands within the spectrum. The power amplifier stage of the off-the-shelf radio chips typically has low linearity to reduce power consumption. This non-linearity results in RF signals at harmonics of the carrier frequency. In a design for a wide spectrum, harmonics of the lower carrier frequencies can lie within the higher frequencies of the spectrum, resulting in spurious RF emissions. As a remedy, our radio design incorporates a collection of electronically switchable RF filters between the SX1262 radio transceiver and the antenna. The filter cut-off frequencies are selected in a way such that, with the appropriate filter selected, the harmonics of any carrier
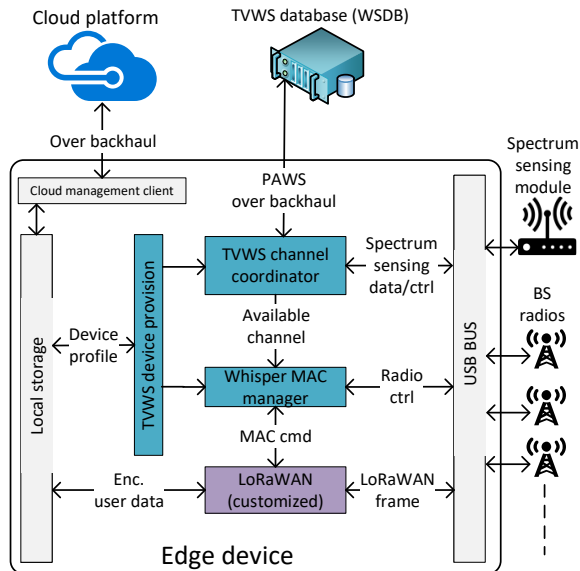
Figure 2: Gateway architecture. Whisper implements it to enable the gateway operation in the TVWS spectrum.

signal within 150 MHz to 960 MHz are filtered out preventing spurious RF emissions. We further carry out laboratory experimentation to ensure that the adjacent channel emissions limits of Whisper radio comply with FCC regulations. Find the setup and results of the experimentation in Appendix A.

Whisper radio has a low-power GPS module to provide its geo-coordinates and height to the WSDB, as required under the FCC rules for unlicensed white space devices (Section 3). To control all of these components and execute our communication protocol, we use an ultra-low-power microcontroller, STM32L151RE based on ARM cortex-M3 architecture [30].

The maximum current consumption by Whisper radio is 119 mA in transmission (at 20 dBm transmit power) and 12 mA in reception with 3.3V power supply. Although the TX energy consumption seems to be higher compared to off-the-shelf ISM band LoRa radio, Whisper radio consumes less energy per given throughput while communicating at a distance of more than a mile. Because the lower frequency of TVWS spectrum enables higher data rate, i.e., less time on air, at a longer distance compared to the ISM band (Section 7.1.4).

4.2 Whisper Gateway

Using the Whisper radio as an IoT client device, we also need a gateway to enable end-to-end communication. Figure 2 shows the architecture of Whisper gateway. It consists of multiple Whisper radios, an edge device, and a spectrum sensing module. First, the radios integrated with the gateway are referred to as base station (BS) radios and follow a similar design to the Whisper radio mentioned above. The number of BS radios can be adjusted depending on the scale of application. Next, the edge device is off-the-shelf can be a single board computer (e.g. Raspberry Pi, Up Board, etc.) or even

a laptop PC. Finally, the spectrum sensing module and BS radios are connected to the edge device through a USB hub.

The edge device has the Whisper MAC manager at its core, which facilitates IoT communication over the TVWS spectrum. It administrates the medium access by the clients (Section 5), coordinates network bootstrap followed by data communication (Section 5.4), and handles the dynamism in TVWS spectrum (Section 6). To do so, it requires exchanging MAC commands with the clients. Here, MAC commands and user data are wrapped in standard LoRaWAN frame format [11]. To be specific, we use the security provided by LoRaWAN along with its frame format, however, not any associated MAC protocol. Consequently, we customize the MAC commands according to our protocol.

TVWS channel coordinator prepares the list of available channels to be used in the network. To do so, it directly communicates with the TVWS database (WSDB) for TV channel availability in the region using the standard protocol to access white space (PAWS) [8]. Furthermore, it conducts real-time screening of the uplink NB channels using the spectrum sensing module (Section 6).

5 Whisper MAC Protocol

With TVWS, we can reduce the number of gateways in a large-area deployment and still support endpoint devices dispersed at varying distances. A question that comes up is, why can we not take a similar approach that is used in mainstream LPWAN communication such as pure or slotted ALOHA? Unfortunately, these protocols do not perform well in our targeted scenarios [1, 14, 16, 26]. In particular, these protocols are not suitable for applications requiring confirmed-up frames (e.g., cameras for plant stress monitoring) and higher traffic load in a single-gateway network (Section 7.2). Besides, a gateway following these protocols appears to be even more inefficient in handling the diversity in data rate demands by the clients [13, 19]. Most importantly, the dynamic nature of the TVWS spectrum is not considered in existing MACs, since these are primarily designed for ISM band operation.

We design and implement FTDMA based MAC protocol to address these challenges. However, note that there are two apparent overheads of FTDMA based protocol for LPWAN: synchronization frame and scheduling control frame. These are neutralized by leveraging the complementary advantages from FCC-mandated compulsory parts of the system. Here, we utilize the onboard GPS module, mounted to comply with FCC regulations detailed in Section 3, for synchronization purposes. Furthermore, we piggyback the scheduling info in the FCC mandated regulatory control frames.

5.1 FTDMA structure

Whisper's MAC protocol has a custom FTDMA structure at its core. We first introduce the structure and corresponding
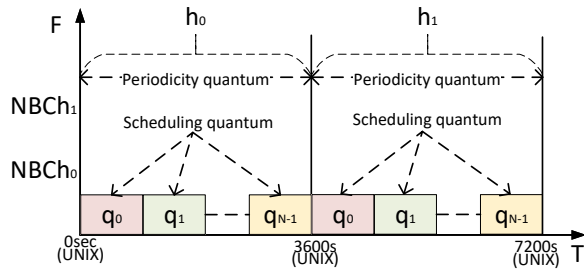
Figure 3: FTDMA structure of Whisper MAC. For communication, each client uses a slot that is a group of consecutive scheduling quanta inside a periodicity quantum.

components. Figure 3 shows a high-level depiction of the proposed FTDMA structure. The T-axis represents time tracked in seconds in the form of a UNIX timestamp. The origin point of the T-axis is starting of the UNIX timestamp which is universal. We divide the T-axis into two dividing time units: periodicity quantum and scheduling quantum. The required periodicity of periodic traffic is basically a multiple of the periodicity quantum. Each periodicity quantum is further divided into an equal number of scheduling quantum. Scheduling quantum (q) is the minimum time precision required in the scheduling. Both periodicity and scheduling quantum can be adjusted based on the computation capability of the edge device and the required precision of scheduling parameters for the application scenario. For the application in our real-world evaluation, we use the hour as the periodicity quantum. To keep the coherence in the rest of the paper, we use hour (h) in place of periodicity quantum. Here, each hour in T-axis is denoted with h_n (Figure 3), where n is the number of hours from the origin point of the T-axis. As mentioned above, we further break each hour down into N scheduling quanta where n^{th} one is denoted with q_n. Now, the F-axis represents the NB channels to be used. Note that if the gateway has multiple BS radios, the T-axis, as well as the set of scheduling quanta, is separate for each radio, however, the F-axis is shared. This allows multiple BS radios to operate simultaneously across different NB channels. The final component of the FTDMA structure is the slot, a group of *consecutive scheduling quanta*. For communication, each client is allocated at least one slot. Note that, in a slot, not more than one uplink (downlink) channel is used. However, multiple uplink (downlink) frames can be transmitted in a slot.

The IoT clients can generate two different patterns of traffic: periodic and event-driven. Here, the timing required for communication depends on the traffic pattern. Depending on the data rate and size of the payload, the time length required for communication varies. We formulate these requirements as the slot requirement in our FTDMA structure. To fulfill the requirement, the slot allocation algorithm of Whisper MAC optimally allocates slots along with communication channels in compliance with the occupancy limit. In the following subsections, we describe the slot allocation algorithm in detail for the two aforementioned traffic patterns.

5.2 Slot allocation for periodic traffic

A client generating periodic traffic requires slots at a regular periodicity. We define the requirement as σ scheduling quanta are required with the periodicity of p, where σ depends on the number of frames, both uplink and downlink, to be communicated and system processing time. One or multiple slots can be allocated having at least σ scheduling quanta in total. Now, the frames to be communicated can have a variable size depending on the data rate and payload size. Therefore, each frame requires at least a certain number of scheduling quanta in a slot for communication. Here, we define continuity (α) as the number of scheduling quanta in a slot. Now, α_{min} is the minimum continuity required for the communication of a frame. Next, we delineate how the slot allocation algorithm of Whisper MAC allocates slots in three phases: scheduling quantum selection, channel selection, and slot allocation.

5.2.1 Scheduling quantum selection

The goal of the scheduling quantum selection process is to find a set of σ scheduling quanta at every p hours which are not a part of the existing allocated slots. As the T-axis is separate for each BS radio, we here describe the quantum selection process for single BS radio. We first define assignment, $A<h_s, p>$, for a scheduling quantum which implies it is occupied at every p hour starting from h_s hour to fulfill a slot requirement. When two assignments of a quantum take place in the same hour, we call it a collision. Consequently, two communication slots corresponding to these assignments collide which is not desirable. Hence, given the new slot requirement, the quantum selection process makes sure that the new assignment for a quantum does not collide with the existing ones. To do so, it leverages the linear Diophantine equation. According to the theorem for solution to linear Diophantine equation, two assignments $A_1<h_{s1}, p_1>$ and $A_2<h_{s2}, p_2>$ collides iff $|h_{s1} - h_{s2}|$ is a multiple of $gcd(p_1, p_2)$. For now, we assume that $p \in \mathbb{Z}^+, 1hr \leq p \leq 24hr$ (cases outside this boundary are discussed in Appendix B.4). Given the boundary of p, $h_s \in \mathbb{Z}^+, 0hr \leq h_s \leq 23hr$.

Now, what if any scheduling quantum is not found collision-free for the new assignment? In this case, we modify the new assignment by making it silent in the hours of collision with existing assignments. As a result, the new client will halt its transmission in the slots of those hours. It is apparent that silencing has an effect on the throughput of the new client, and thus, it should be minimum. Here, we introduce a metric $\omega_{q\%}$ that measures the frequency of silencing. It can be calculated from the generic solution of the Diophantine equation. See Appendix B.1 for more details.

The scheduling quantum selection process tunes the value of h_s from h_0 to $h_{(p-1)}$ for the required periodicity p and makes the list of collision-free scheduling quanta with and without modifying the new assignment to be silent. For the slot selection algorithm, it sets a priority value ($\rho_q \in \mathbb{R}, 0 \leq$
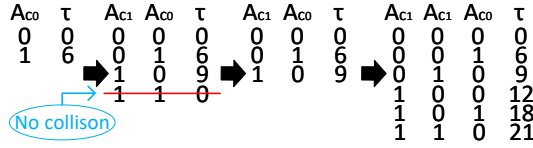
A_c0 τ | A_c1 A_c0 τ | A_c1 A_c0 τ | A_c1 A_c1 A_c0 τ

A_{c0} τ
0 0
1 6
→ 1 0 9
1 1 0 (No collison)

A_{c1} A_{c0} τ
0 0 0
0 1 6
1 0 9
→

A_{c1} A_{c0} τ
0 0 0
0 1 6
1 0 9
→

A_{c1} A_{c1} A_{c0} τ
0 0 0 0
0 0 1 6
0 1 0 9
1 0 0 12
1 0 1 18
1 1 0 21

Figure 4: Construction of channel occupancy table. Whisper uses it to formulate and optimally limit the channel occupancy.

$\rho_q \leq 1$) to every scheduling quantum in the list as following.

$$\rho_q = \begin{cases} 1, & \text{has old } A[], \text{ no collision with new } A \\ 2/3, & \text{does not have assignment} \\ \frac{1-Max(\omega_{q\%})}{3}, & \text{has old } A[] \text{ and collides with new } A \end{cases}$$

A scheduling quantum having existing assignments and no collision with the new assignment gets the higher priority compared to the one with no existing assignment. It ensures optimal usage of a quantum by grouping non-colliding assignments together. Finally, a quantum having existing assignments colliding with the new one gets the lowest priority depending on the frequency of silence.

5.2.2 Channel selection

The channel selection process, completely independent of quantum selection, optimally finds the channels for new slot requirement complying with the occupancy limit. First off, if two assignments use the same channel in the same hour, it is a collision in channel usage. However, the collision is safe, i.e., assignments are valid, as long as the total occupancy of the channel in that hour is not more than 36s (Section 3). Based on it, validating a new assignment for a downlink channel is challenging. A downlink channel might be assigned to multiple clients, and these assignments can collide in different combinations and hours. Furthermore, in a gateway having multiple BS radios, although the T-axis is separate for each BS radio, the F-axis is shared, and the channel occupancy is calculated in aggregate for all BS radios. Now, simply avoiding the collision in channel usage by making the assignment silent in the colliding hour or using spare channels result in significant throughput loss and wastage of bandwidth respectively. Therefore, we need to formulate the channel occupancy and optimally limit the same despite collision among assignments in channel usage. Here, we propose binary counting based dynamic table on top of the linear Diophantine equation to validate an assignment for a downlink channel.

Now, we illustrate the dynamic construction of channel occupancy table with Figure 4. We here define the assignment for channel as $A_c<A, \tau, q[]>$, where τ denotes the channel occupancy (in seconds) per hour for the assignment, and $q[]$ denotes the list of scheduling quanta associated with it. $q[]$ is used in tracking the channel overlap in FDMA. As shown in Figure 4, each entry in the table contains the colliding assignments and corresponding total occupancy (τ) in the colliding hour of the assignments. For example, an entry $<A_{c3} = 1, A_{c2} = 0, A_{c1} = 1, A_{c0} = 1, \tau = 17>$ means A_{c3}, A_{c1}, and A_{c0} collide in channel usage, and the channel occupancy

is 17s in the colliding hour. Now, if there are n assignments for a channel, there are 2^n possible combinations of assignments. However, the assignments in a possible combination may not collide, and thus it can be ruled out for future computation. For example, A_{c0} and A_{c1} do not collide, and thus the last entry in the second table has $\tau = 0$ that is removed for any further development. Find step by step description of table construction with example in Appendix B.2

The channel selection process leverages the channel occupancy table in finding a valid channel despite the collision between new and existing assignments in an hour. However, the question arises which channel is optimal to select? To do so, we here utilize a priority variable (ρ_c) for a channel similar to (ρ_q). However, $\omega_{c\%}$ is treated differently compared to $\omega_{q\%}$. In case of a collision between the new and existing assignments in channel usage, although the occupancy becomes higher only in the hour of colliding occurrence, the channel remains under-utilized in the rest of the hours of non-colliding occurrences. Consequently, a higher value of $\omega_{c\%}$ results in lesser under-utilized hours. Hence, instead of $\frac{1-Max(\omega_{q\%})}{3}$, we use $\frac{Avg(\omega_{c\%})}{3}$ in calculating ρ_c for a channel. Although it cannot ensure complete eradication of under-utilized hours for a channel, we later show how the channel can be utilized further in these hours for event-driven traffic. In aggregate, for a given assignment A_c, the channel selection process finds a valid downlink channel using the occupancy table with the maximum possible ρ_c.

5.2.3 Slot allocation

The slot requirement from a client can be satisfied with a single or a combination of slots. From a high level, the slot allocation algorithm works in two steps. First, it prepares different possible combinations of slots using the scheduling quantum selection and downlink channel selection (find the pseudo code in Appendix B.3). Then, it selects the best combination of slots based on the following weight function.

$$f = w_1\rho_q + w_2\rho_c + w_3\alpha - w_4\psi$$

We have discussed ρ_q and ρ_c earlier. Positive weight on higher continuity (α) value takes lesser slots to fulfill the requirement of σ quanta, which, in turn, saves the power of the client. On the other hand, ψ quantifies the wastage of quanta, i.e., how many extra quanta are used in total for a combination of slots. Consequently, a negative weight on ψ reduces the wastage in bandwidth-sensitive applications. All weights can be further tuned based on the nature of the application.

5.3 Slot allocation for event-driven traffic

In event-driven traffic, a frame is generated based on an event-trigger, e.g., rain monitoring. The frame needs to reach the gateway before a certain time. Therefore, we define the requirement as σ scheduling quanta are required with α_{min} con-
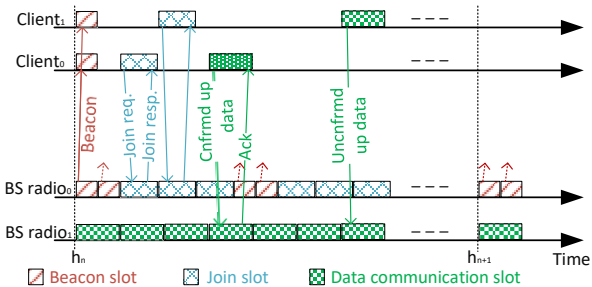
Figure 5: Time sequence diagram of Whisper protocol in a simple example scenario. This example depicts bootstrapping of two clients followed by data communication with the gateway having two BS radios.

tinuity before the expiry, ε. The underneath concept of slot allocation algorithm for event-driven traffic is mostly similar to the periodic traffic. We point out the *distinct factors* below.

During scheduling quantum selection, we check the collision of an assignment for event-driven traffic with the existing assignments for both types of traffic. However, there is no notion of silencing the new assignment for event-driven traffic as its time-sensitive. Accordingly, ρ_q has three different priority levels - highest priority to no collision with existing assignments, then to no existing assignments, and lowest to collision only with existing assignments for periodic traffic. Here, in the last case, even if the existing periodic assignment ends up transmitting frames in the same channel together with the new event-driven assignment, according to the nature of LoRa physical layer, there is a probability of one getting successfully demodulated. In this way, the scheduling quantum selection process enlists all quanta before the expiry, ε.

Although the underneath concept remains similar, the channel selection for event-driven traffic is more simplistic than periodic traffic. Here, rather than using the channel occupancy table, the validity of a new assignment for a channel is probed by simply checking collision with individual existing assignment for both types of traffic in the hour of the new assignment. If the new assignment collides with existing assignments in channel usage and the total τ of all colliding assignments including the new one remains within the limit, we consider it valid. The priority of a channel (ρ_c) for a valid new assignment is decided differently – highest priority to no collision with existing assignments, then to collision with existing assignments, and lowest to no existing assignments. It facilitates the utilization of a channel in no or low utilization hours of the periodic assignments.

Finally, we make a modification in the weight function for slot allocation as following.
$$g = w_1\rho_q + w_2\rho_c + w_3\alpha - w_4\psi - w_5\delta$$
where δ is the time difference between selected scheduling quantum and the expiry (ε). A negative weight on δ makes the choice for immediate slot less greedy. It, in turn, facilitates serving multiple concurrent requests of event-driven traffic without colliding in the immediate slot.

5.4 Bootstrap and Data Communication

Hitherto we discuss the process of slot allocation for controlling the medium access by the clients. We now describe how Whisper MAC manages client bootstrapping and data communication using the slot allocation. Client bootstrapping is a critical part of a TVWS based IoT network since it is required to comply with FCC regulations given the power constraint of the IoT devices. Recall that, each NB TVWS device must provide the WSDB its geo-coordinates and height to obtain the list of available channels for transmission at that location. Here, the clients can register via the gateway as they do not have Internet connectivity. Nevertheless, a key question remains. How does a client share its location with the gateway without knowing the available channel and time for transmitting registration request?

According to the FCC regulations, a client that is not registered in the WSDB can transmit only its location information or network join request on the channels registered against the gateway. To do so, a client requires to know the channels registered against the gateway where the list of these channels may even vary over time. As a remedy, Whisper MAC utilizes broadcast beacons that embed the info of join slots for clients to transmit WSDB registration, i.e., network join request. In the join slot, a client transmits the join request along with its location for WSDB registration and slot requirement for the data communication. Note that the beacon slots have predefined and fixed timing and channels. Therefore, the time synced (using GPS) clients can listen for the beacon frames without draining its power in random or continuous scanning. Figure 5 depicts an example scenario of client bootstrapping followed by data communication. Due to space limitations, we put the corresponding implementation details in Appendix B.5.

After receiving the WSDB registration response, i.e., join response, a client is ready for the data transmission in the allocated slots (piggybacked as MAC command in the response frame) until the expiry of channel. A TVWS device is required to contact (poll) the WSDB once every 24 hours and check the list of available channels at the location. The poll request is piggybacked as a MAC command in a confirmed-up data frame. The expiry generally gets extended after the polling, and updated expiry is sent in the downlink ACK. However, polling is not required for a client generating event-driven traffic as it freshly joins the network whenever it has a data frame to transmit.

6 Dynamic Spectrum Access

The dynamic nature of TVWS spectrum can lead to bottlenecks when it is utilized for NB IoT networks. First, the long distance between the gateway and a client brings about separate uplink and downlink having dissimilar quality at their corresponding locations. Second, the IoT clients are power
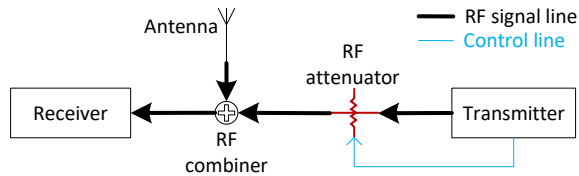
Figure 6: Block diagram of our spectrum sensing module that measures interference from the incumbent users.

constrained. Hence, any extra transmission and carrier sensing activity by the clients for spectrum management adds power overhead. Whisper addresses these challenges by implementing spectrum awareness capability and then incorporating smart spectrum exploitation approach in the protocol.

6.1 Spectrum awareness

Uplink communication is inevitably significant in most of the IoT applications as IoT traffic is push-based. The quality of this communication link directly relies on the interference nearby the gateway. Hence, we first try to enable spectrum awareness at the gateway. A part of spectrum awareness, both uplink and downlink, is achieved by using the WSDB. However, it is not enough to accumulate info about the real-time activity of incumbent users – both licensed and unlicensed – in the carrier. To do so, Whisper performs spectrum sensing.

First off, how can the interference level in a NB channel be measured given the Whisper radio uses LoRa modulation and its minimum detectable signal (MDS) is very low (-149 dBm)? The LoRa modulation enables frame reception even when the carrier signal level is below the ambient RF noise floor. Consequently, the conventional approach of measuring RF energy in a NB channel as an indication of interference level is not viable here. Instead, we design a custom off-the-shelf spectrum sensing module to evaluate the level of interference in NB TVWS channels. It can evaluate the impact of interference from licensed and unlicensed cognitive radio users at a very low signal level.

Spectrum sensing module locally generates RF signal with a controlled amplitude to determine MDS in a NB channel. The MDS gives the perception of interference from incumbent users in the NB channel – a lower MDS implies a lower RF interference. Figure 6 depicts the block diagram of the spectrum sensing module. It consists of a transmitter, a controlled attenuator, an RF combiner, and a receiver. We use two Whisper radios as the transmitter and receiver. The attenuator is controlled by the transmitter. We modify transmitter radio firmware to set the value of attenuator. To estimate the MDS in a channel, we tune the transmitter and receiver to the specific channel frequency, and the transmitter starts transmitting test frames. The RF signal of a test frame is attenuated by the attenuator to yield a weak signal similar to a reception from a remote transmitter, which is a commonly practised technique in laboratory emulation [15]. The combiner combines this weak signal with the ambient RF signals (includes interfer-

ence from incumbent users) picked up by the antenna. The attenuator value is varied to identify the maximum attenuation that results in 90% successful frame reception. The lowest RSSI of the received frames is the MDS under the interference. Note that the spectrum sensing module only senses the RF channel. The generated RF signal is only used internally, and thus has no effect on data communication and channel occupancy limit. The spectrum sensing module is a part of the gateway as mentioned in Section 4. The channel control unit utilizes it every hour to update the interference level from the incumbent users.

6.2 Spectrum exploitation

The interference in the downlink at the location of a client cannot be measured with the spectrum sensing module, given the long distance. Furthermore, carrier sensing at the client's end would add huge power overhead. Here, Whisper enables the spectrum exploitation approach to mitigate the effect of interference through dynamic channel assignment that does not require any additional frame transmission.

Before jumping into the details, we first introduce the related terminologies and notations. For each slot, we define a channel-tube (ChT) that has two channels for a confirmed-up and one channel for an unconfirmed-up data frame. For a slot used in periodic traffic, N_{ChT} channel-tubes are assigned by adjusting the A_c for the associated channels during the slot allocation described in Section 5. In every occurrence of the slot, a ChT is picked in a sequential round-robin manner for communication. In case of event-driven traffic, a ChT is assigned which is different from last $(N_{ChT} - 1)$ ChTs. In this way, when a slot completes hopping across all N_{ChT} ChTs, we call it a hopping cycle. Note that, a client may have multiple slots (as mentioned in Section 5) and each slot has an individual set of N_{ChT} ChTs. Furthermore, ChTs are preferably selected from different TV channels to facilitate robustness against broadband interference. Next, we delineate how the set of ChT is dynamically updated in three phases.

1. Monitoring phase: During this phase, the performance of assigned ChTs for all the slots of a client is monitored at the gateway, and a flag is raised in case of substandard performance. It is raised based on the overflow of a bucket that is filled up with the tickets for an event of missing frame in a ChT. The number of tickets for a missing event is equal to the number of previous consecutive occurrences starting from the current event in the ChT. In the event of successful communication in a ChT, we remove its tickets in a similar manner. Now, how do we detect an incident of the missing frame? We here leverage the frame count block of the standard LoRaWAN frame header. Finally, we need to determine the optimal bucket size (S_{bukt}). It is directly related to $\sum_{i=1}^{ND_{ChT}} k_i$ where ND_{ChT} is number of distinct ChTs and k is number of times it is used in a cycle. S_{bukt} depends on the length of hopping cycle too. For example, a client with a smaller cycle

overflows the bucket faster in presence of a short-term interference, whereas it gets slower in case of a longer cycle with long-term interference. Both these scenarios are undesired. Here, we express S_{bukt} as $m\sum_{i=1}^{ND_{ChT}} k_i$ for different bands of cycle. From our emulation and real-world deployments, we found the following optimal values of m with $N_{ChT} = 4$ for each slot. Here, m typically exhibits an exponential downward pattern with respect to the length of cycle.

$$m = \begin{cases} 2.5, & \text{cycle length} < 1hr \\ 1.6, & 1hr \leq \text{cycle length} < 3hr \\ 1.05, & 3hr \leq \text{cycle length} < 6hr \\ 0.7, & 6hr \leq \text{cycle length} < 11hr \\ 0.5, & 11hr \leq \text{cycle length} \end{cases}$$

2. Decision making phase: After a flag has been raised, the decision making phase decides which ChT and associated channels are to be replaced. We note that a ChT is selected for replacement if its contribution in bucket overflow is at least a certain percentage defined as, $\frac{k_i}{\sum_{i=1}^{ND_{ChT}} k_i}$. If no such ChT is found, older events are removed to accommodate new events in the bucket. Next, we find which channel(s) in the ChT(s) needs to be replaced based on the knowledge from the monitoring phase. The replacement ChT is placed exactly at the same sequential position of the replaced one in the current list of ChTs for a slot.

3. Execution phase: For a client sending confirmed-up frames, the execution of ChT update is initiated immediately after the decision making. Whereas, in case of a client sending unconfirmed-up data frames, it is carried out when the client polls the channel from WSDB since this is the only time when the client sends confirmed-up data frames. The execution is initiated by the gateway by sending an update notice in the downlink frame. The client sends a notification of notice reception in the following uplink data frame. To ensure the robustness against frame loss, the update notice (if a confirmed up frame is received) and notification are sent in the following N_{ChT} slots. The timing of the update is set accordingly in the notice. Note that the execution process is separate for each slot. Furthermore, no extra frame is exchanged for the execution process since all the associated MAC commands are piggybacked in the data frames.

6.3 Fallback

Finally, in a rare event of complete communication loss with the gateway, a client calls for the fallback. Although a client sending confirmed-up data frames detects such an event straight away, a client sending unconfirmed-up data frames detects only at the time of channel polling from WSDB. In such an event, the client does not get a downlink packet in any slot and ChT. As a fallback alternative, the client rejoins the network. It reports the event in the join request so that gateway can assign a different set of ChTs.

	Avg.	Max	Min
Throughput (in bps)	0.0626	0.0681	0.0601
FDR (in %)	98.4	100	97.1
Latency (in sec)	2.39	2.43	2.37

Table 2: Performance of Whisper in real-world deployment

7 Evaluation

In this section, we focus on evaluating the performance of Whisper through a real-world deployment and simulation. We further make a comparison with the ISM band IoT solutions.

7.1 Real-world deployment

We have an ongoing deployment of Whisper on 8500 acre dry-land wheat farm in Eastern Washington, which has been operating for over 2.5 months. Here, Whisper is used by the third-party users in collecting sensor data from 17 different fields for multiple data-driven agriculture applications.

7.1.1 Setup

The deployment includes 20 Whisper client radios that communicate with a single gateway. The client radios are retrofitted inside a weatherproof box, powered by solar, and connected to a sensor interface to collect data from five sensors – temperature, humidity, CO_2, and soil moisture and temperature (Figure 7b). As configured by third-party users, 11 clients report data of all sensors at different periodicity starting from 30 min to 12hr. Similarly, the remaining clients report all sensor data in an event-driven manner based on CO_2 level. All of the data frames are confirmed-up – 51 bytes uplink, 20 bytes downlink. The modulation parameters are configured to have a coding rate (CR) of 4, 62.5 kHz channel bandwidth (ChBW), and preamble length of 8. We tune the spreading factor (SF) based on the distance of the clients from the gateway as shown in Figure 7c.

The gateway is deployed on the farm, and due to a lack of power, is powered by solar power (Figure 7a). Since there is no Internet connectivity in the middle of the farm, we use an Adaptrum TVWS broadband radio at the gateway to create a wireless link to the nearest source of connectivity (farmer's home) [2]. It in turn enables the evaluation of Whisper in coexistence with off-the-shelf unlicensed TVWS broadband network. We use a Raspberry Pi 3B as the edge device at the gateway. For WSDB, we use Wave DB Connect by RED Technologies [31]. Finally, both BS radio and clients use omnidirectional antennas with approximately 5 dBi gain where the TX power is set to the max according to Section 3.

7.1.2 Results

Thus far, 4766 sensor data points have been collected via Whisper. We evaluate the performance of Whisper in terms of three metrics: throughput, frame delivery ratio (FDR), and
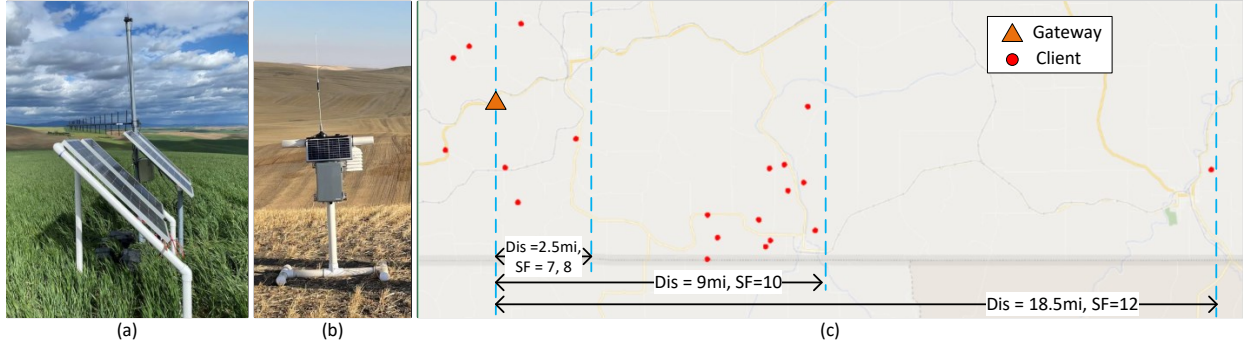
Figure 7: Whisper Deployment. The solar powered IoT Hub shown in (a) includes a broadband TVWS backhaul and Whisper gateway. In (b), Whisper client integrated with sensors and retrofitted inside a weatherproof box. (c) shows the deployment map.
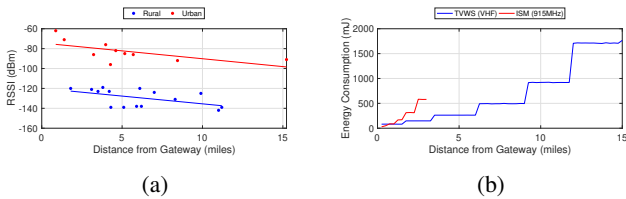

Figure 8: (a) Range of Whisper radio and (b) Comparison of energy consumption between Whisper radio in TVWS VHF band and off-the-shelf LoRa radio in ISM band.

latency. The value of each metric is averaged over a period of 24hr. We do not observe any significant deviation in the average values of these metrics throughout the deployment period. Table 2 summarizes the results. We also observe that Whisper has shown robustness against multiple real-world incidents such as power outages at gateway and clients due to thunderstorms, disrupted Internet connectivity, and WSDB disruption due to maintenance. Furthermore, Whisper shares the spectrum (470 to 488 MHz) with Adaptrum broadband TVWS radio in the deployment as we mentioned above. However, we do not observe any mutual interruption in the communication of Adaptrum radio and Whisper.

7.1.3 Range

We perform range experiments to evaluate the sensitivity of the Whisper radio. Range tests are conducted in both urban and rural environments, where both have line-of-site (LOS) and non-line-of-sight (NLOS) connectivity. For the urban settings, we place a BS radio on the rooftop of an industry campus building. Then, we move the client to different locations using a vehicle and record the RSSI. The radios are configured for LoRa modulation having a ChBW of 31.25 kHz, SF of 12, and CR of 4 in the VHF band. Figure 8a shows the RSSI of received frames with respect to distance in miles. We can see that frames are successfully received at 11.3 miles with RSSI of -85 dBm. For the evaluation in rural settings, we use our aforementioned farm deployment. In this scenario, the performance is even better in comparison to the urban settings. We can see that packets are successfully received at

15 miles with a measured RSSI of -87 dBm which is significantly higher than the MDS (-149 dBm) of Whisper radio. These results imply that clients can be deployed as far as 15 miles away from the gateway, and likely even further. For example, in Figure 7c, one of the clients communicates with the gateway from a distance of 18.5 miles. Note that, the range of LoRa in ISM band is found to be 1 - 3 miles in literature and our real-world deployment [4].

7.1.4 Energy profile

We next conduct energy profiling of Whisper radio operating in TVWS spectrum and compare it with an off-the-shelf LoRa radio, SX126xMB2xAS [20], operating in the US915 ISM band. The physical setup for the experimentation is similar to the range test. Both radios are configured to send an uplink frame of 51 bytes every hour to the gateway with the same LoRa modulation configurations (CR: 1, ChBW: 62.5 kHz, TX power: 20 dBm). We then vary the distance of the radios from the gateway and tune the SF accordingly for successful communication. In Figure 8b, we show the energy consumption for each frame transmission in TVWS VHF band (174.3 MHz) and ISM band (915 MHz). Note that the energy consumption of the Whisper radio in locking GPS (once in 24hr) is prorated over its transmitted frames. Up to the initial distance of around a mile, the energy consumption in the TVWS spectrum is higher. However, as the distance increases, the time on the air for frame transmitted in the ISM band increases due to the higher spreading factor. Whereas, the lower frequency of the VHF band facilitates longer distance with comparatively lower SF. Consequently, the energy consumption becomes at least 2x higher in the ISM band compared to the TVWS VHF band after a mile of distance.

7.1.5 Performance in presence of interference

To evaluate the performance of Whisper under interference in the above-mentioned real-world deployment, we create interference with a separate transmitter. The interference is introduced on both uplink and downlink channels in three different ways - intermittent, continuous, and complete block.
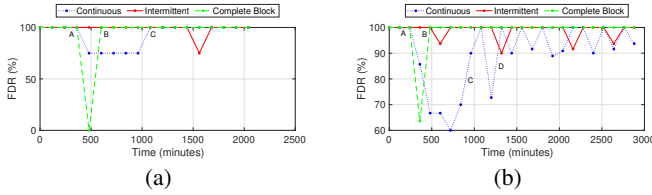
Figure 9: Whisper's response to different types of interference on (a) downlink and (b) uplink channels in real-world.

In the case of intermittent interference, the interferer hops across the channels of $N_{ChT} = 4$ *ChT*s and creates short-term interference for the period equals to the time on air of a frame. Whereas, in continuous interference, the interferer picks one of the channels from the 4 *ChT*s and creates continuous interference on that channel. Finally, in case of complete block, the interferer is time-synced and aware of the hoping cycle schedule of the target radio. It accordingly creates interference on the active channel for data communication.

We evaluate the impact of these three types of interference on downlink channels. The interferer is placed very close to a client with high transmit power. The interferer is aware of the downlink channels used by the client. We then record the FDR on the downlink channels and the activity of switching channels for the client. In this way, we investigate the impact of interference for the clients, having different periodicity and traffic patterns, in our aforementioned deployment. Since all the clients generating periodic traffic exhibit similar behavior, we only discuss a client having a periodicity of 30 min. Figure 9a shows downlink FDR of the client averaged over a period of 120 min (one hopping cycle). As shown in the figure, the intermittent interference has a very minor impact on the FDR. No *ChT* update is initiated in response, whereas, for continuous interference, we observe a significant drop in FDR after the interferer is activated (point A in Figure 9a). As a result, a *ChT* update is initiated by the gateway. After the execution of same, the FDR reaches 100% (point C in Figure 9a). Finally, for complete block interference, we see the complete cease in downlink communication for the client. Consequently, the client triggers the fallback mechanism and rejoins the network (point B in Figure 9a). As a new set of *ChT*s is assigned after rejoining, the FDR reaches the maximum. We observe a similar impact of intermittent interference and a lesser impact of continuous interference on clients generating event-driven traffic. This is due to the members of the set of 4 *ChT*s varying with time. Nevertheless, it is difficult to create complete block interference for event-driven traffic.

Next, we evaluate the impact on uplink channels by placing the interferer close to the gateway. We select four uplink channels of the same client for the interferer, however, these channels are used by more than half of the clients in different combinations. To get a precise understanding of the effect of interference, we record uplink frames transmitted by any client on only these four channels. Figure 9b shows corresponding uplink FDR averaged over a period of 120 min. We

observe very minimal effect of the intermittent interference on the network and no *ChT* update took place for the same. In case of continuous interference, we see a drop in the FDR after the activation of interferer (point A in Figure 9b). Over the period of two days, *ChT*s of two clients are updated followed by increment in FDR (point C and D). Besides, the spectrum sensing unit detects the interference on the channel, and consequently, the channel is not further assigned to any client for sending event-driven traffic in this period by the channel coordinator. Finally, the complete block interference only affects the communication of one client. The client exhibits similar response as described above in case of the downlink.

7.2 Simulation

We now focus on evaluating Whisper's capability in terms of scale. Since deploying a very large-scale network (100s of clients) is challenging, we carry out the simulation. In particular, we evaluate two things: (1) how the larger bandwidth of TVWS spectrum facilitates scaling and (2) how Whisper MAC makes better utilization of this bandwidth compared to mainstream IoT MAC protocols. For this purpose, IoT devices transmitting images is a compelling application scenario (Section 2). We simulate this scenario in three network setups: (S1) Whisper MAC in TVWS spectrum, (S2) Whisper MAC in ISM band, and (S3) pure ALOHA (used by LoRaWAN) in TVWS spectrum. Comparison between S1 and S2 shows the role of bandwidth in our application scenario, whereas S1 and S3 show how Whisper MAC makes better utilization of this larger bandwidth.

7.2.1 Setup

We first integrate the LoRa physical layer from FLoRa simulator and the simulation environment of OMNet++ with our protocol [29]. We set up a single-gateway star-topology network where every IoT client device sends a 25 kB image. The gateway has eight BS radios. The number of clients is varied from 5 to 1000 at an increment of 5. The ratio of clients generating periodic traffic (N_p) to event-driven traffic (N_e) is also adjusted in five levels. We make an even distribution of image generation periodicity starting from 1hr to 24hr among the clients generating periodic traffic. A client with event-driven traffic sends an image once in every 24hr following random distribution. For S1 and S3, the uplink frame size is 255 bytes and downlink ACK frame is 18 bytes. The LoRa modulation parameters are set as following - CR: 1 and ChBW: 62.5 kHz. To simulate the effect of distance and associated data rates, we make an even distribution of six possible SF values ($7 - 12$) among the clients. Beacon periods are separated by two minutes with a periodicity of $p_{bcn} = 1$hr (Appendix B.5). For simulation purposes, since radios are not certified real devices, we use a local proxy of WSDB where each device, including gateway, has 3 TV channels available for transmis-

(a) Throughput

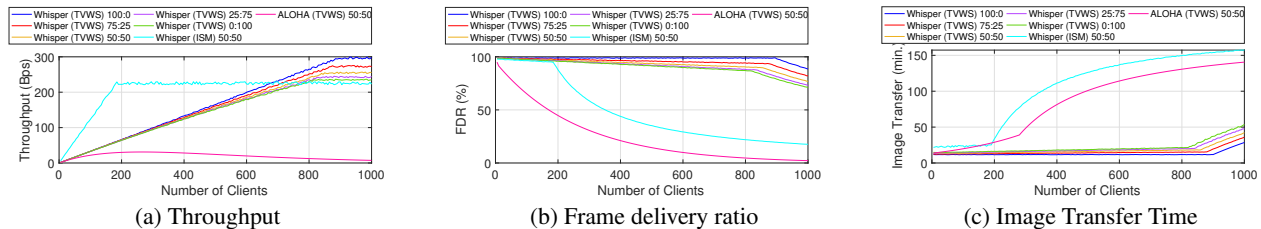(b) Frame delivery ratio

(c) Image Transfer Time

Figure 10: Simulation results showing the role of large TVWS band and how Whisper MAC better utilizes the same in scaling.

sion. For S2, we use the four uplink data rates, DR0-DR3, permitted in the US915 ISM band where corresponding SF ranges from 10 to 7 with the channel bandwidth of 125 kHz. The dwell time restriction limits the uplink payload size (in bytes) for these data rates – DR0:11, DR1:53, DR2:125, and DR3:242. On the downlink side, we use data rates, DR10-DR13, having same SFs as uplink with ChBW of 500 kHz.

7.2.2 Results

We analyze the performance in terms of throughput, FDR, and image transfer completion time.

1. Throughput: As shown in Figure 10a, we analyze the throughput of the network under a various number of clients and ratios of traffic types. Each point in the graph represents the average throughput of the network in a period of 24hr. Here, we only consider the data frames, not any frames associated with the bootstrapping process. For S1 (Whisper in TVWS), the throughput increases linearly with the number of clients for every ratio of traffic types ($N_p : N_e$) until it reaches the network capacity. As the proportion of event-driven traffic increases in a group of clients, the throughput slightly drops due to the higher collision rate among the randomly triggered events. For S2 (Whisper in ISM band) and S3 (ALOHA in TVWS), we only show the representative results from the traffic ratio of 50 : 50. Although S2 shows similar patterns as S1, there are two distinct factors. First, due to the limited bandwidth in ISM band compared to the TVWS spectrum, S2 reaches the saturation point with ∼ 5x fewer clients than S1. Second, for the same number of clients in the network, S2 exhibits higher throughput as per-channel bandwidth is higher and supported SF is lower in the US915 ISM band compared to the TVWS spectrum. However, the lower value of supported SFs in the ISM band results in a much shorter range. In case of S3, both maximum throughput and number of supported clients are significantly lower than S1. This shows why the existing mainstream IoT MAC protocol is not suited for making better utilization of large TVWS bandwidth.

2. FDR: In computing FDR, we consider the frame generated but not successfully transmitted, even due to implied silence by Whisper MAC, as a frame loss. Figure 10b provides similar insights as the throughput. Although the initial throughput is found higher in S2 than S1, FDR is lower in the ISM band due to the larger number of small frame (both uplink and downlink) transmission for sending each image

(Section 2). It, in turn, increases the probability of frame loss.

3. Image transfer completion time: We further report the time required to send an image including the re-transmission of frames. The image is dropped after 3hr, if it is not completely transmitted. As shown in Figure 10c, the completion time linearly increases for S1 and S2 until it hits the saturation point, whereas it increases exponentially for S3. Now, although S2 exhibits higher throughput than S1 at the cost of a much shorter range, S2 reports at least 2x higher completion time. Because, in ISM band, a client requires to transmit a larger number of small frames to send an image due to the cap on uplink payload size (Section 2).

8 Related Work

Even though unlicensed operations in TVWS spectrum have been extensively studied in literature [5,6,21]. Nearly all prior work have focused on broadband scenarios and corresponding FCC regulations. However, there are fundamental differences in NB operation such as power constraint of IoT devices, number of devices in the network, channel occupancy limitation, etc. These make the challenges in unlicensed NB operation non-trivial to solve. There are some prior work on devising suitable signal modulation techniques for NB operation in TVWS spectrum [25, 27, 28]. However, dynamic spectrum access and management in NB operation, compliance with the NB FCC regulations, and corresponding comprehensive MAC protocol design are beyond the scope of these work.

9 Conclusion

In this paper, we have presented Whisper, a new solution to enable long-range and wider spectrum communication for IoT by leveraging TVWS spectrum. With Whisper, we get at least 5x longer range compared to LoRa operating in ISM band. Consequently, in our real-world deployment, Whisper has covered 17 fields in a 8500 acre farm with single gateway. Besides, the lower frequency of TVWS spectrum facilitates 2x less energy consumption than ISM band at a range of more than a mile. Note that, for a deployment spanning not more than a mile from the gateway, LoRa in ISM band would be preferable over TVWS spectrum. It also holds for an indoor scenario. Finally, with only 3 white TV channels, Whisper can accommodate 5x more traffic than ISM band.

References

[1] Khaled Q Abdelfadeel, Dimitrios Zorbas, Victor Cionca, and Dirk Pesch. *FREE* — Fine-grained scheduling for reliable and energy-efficient data collection in LoRaWAN. *IEEE Internet of Things Journal*, 7(1):669–683, 2019.

[2] Adaptrum. Adaptrum TVWS broadband radio, 2021. https://www.adaptrum.com/Content/docs/acrs2_datasheet_1016.pdf.

[3] Arjan. Airtime calculator for LoRaWAN, 2021. https://avbentem.github.io/airtime-calculator/ttn/us915-dl/38.

[4] A. Augustin, J. Yi, T. Clausen, and W.M. Townsley. A study of LoRa: Long range low power networks for the Internet of Things. *Sensors*, 2016.

[5] Paramvir Bahl, Ranveer Chandra, Thomas Moscibroda, Rohan Murty, and Matt Welsh. White space networking with Wi-Fi like connectivity. *ACM SIGCOMM Computer Communication Review*, 39(4):27–38, 2009.

[6] Ranveer Chandra, Thomas Moscibroda, Paramvir Bahl, Rohan Murty, George Nychis, and Xiaohui Wang. A campus-wide testbed over the TV White Spaces. *SIGMOBILE Mob. Comput. Commun. Rev.*, 15(3):2–9, November 2011.

[7] Tonghao Chen, Derek Eager, and Dwight Makaroff. Efficient image transmission using LoRa technology in agricultural monitoring IoT systems. In *2019 International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 937–944. IEEE, 2019.

[8] V Chen, S Das, L Zhu, J Malyar, and P McCann. Protocol to access white-space (PAWS) databases. *Internet Engineering Task Force (IETF)*, 2015.

[9] Federal Communications Commission. FCC rules for unlicensed white space devices. March 2020.

[10] Federal Communications Commission. Unlicensed white space device operations in the television bands. *FCC ET Docket No. 20-36, Report Order and Further Notice of Proposed Rulemaking*, October 2020.

[11] LoRa Alliance Technical Committee. LoRaWAN 1.0.3 specification, 2018. https://lora-alliance.org/sites/default/files/2018-07/lorawan1.0.3.pdf.

[12] Semtech Corporation. Semtech SX1262, 2020. https://www.semtech.com/products/wireless-rf/lora-transceivers/sx1262.

[13] Joseph Finnegan, Ronan Farrell, and Stephen Brown. Analysis and enhancement of the LoRaWAN adaptive data rate scheme. *IEEE Internet of Things Journal*, 7(8):7171–7180, 2020.

[14] Jetmir Haxhibeqiri, Ingrid Moerman, and Jeroen Hoebeke. Low overhead scheduling of LoRa transmissions for improved scalability. *IEEE Internet of Things Journal*, 6(2):3097–3109, 2018.

[15] Jetmir Haxhibeqiri, Floris Van den Abeele, Ingrid Moerman, and Jeroen Hoebeke. LoRa scalability: A simulation model based on interference measurements. *Sensors*, 17(6):1193, 2017.

[16] Md Tamzeed Islam, Bashima Islam, and Shahriar Nirjon. Duty-cycle-aware real-time scheduling of wireless links in low power WANs. In *2018 14th International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pages 53–60. IEEE, 2018.

[17] Akram H Jebril, Aduwati Sali, Alyani Ismail, and Mohd Fadlee A Rasid. Overcoming limitations of LoRa physical layer in image transmission. *Sensors*, 18(10):3257, 2018.

[18] Mookeun Ji, Juyeon Yoon, Jeongwoo Choo, Minki Jang, and Anthony Smith. LoRa-based visual monitoring scheme for agriculture IoT. In *2019 IEEE Sensors Applications Symposium (SAS)*, pages 1–6. IEEE, 2019.

[19] Shengyang Li, Usman Raza, and Aftab Khan. How agile is the adaptive data rate mechanism of LoRaWAN? In *2018 IEEE Global Communications Conference (GLOBECOM)*, pages 206–212. IEEE, 2018.

[20] ARM MBED. SX126xMB2xAS, 2021. https://os.mbed.com/components/SX126xMB2xAS/.

[21] Rohan Murty, Ranveer Chandra, Thomas Moscibroda, and Paramvir Bahl. Senseless: A database-driven white spaces network. *IEEE Transactions on Mobile Computing*, 11(2):189–203, 2011.

[22] The Things Network. Fair use policy explained, 2021. https://www.thethingsnetwork.org/forum/t/fair-use-policy-explained/1300.

[23] The Things Network. Regional parameters, 2021. https://www.thethingsnetwork.org/docs/lorawan/regional-parameters/.

[24] NCSU College of Agriculture and Life Sciences. Low-cost cameras could be sensors to remotely monitor crop stress, 2022. https://shorturl.at/evxGW.

[25] Mahbubur Rahman, Dali Ismail, Venkata P Modekurthy, and Abusayeed Saifullah. LPWAN in the TV White Spaces: A practical implementation and deployment experiences. *arXiv preprint arXiv:2102.00302*, 2021.

[26] Brecht Reynders, Qing Wang, Pere Tuset-Peiro, Xavier Vilajosana, and Sofie Pollin. Improving reliability and scalability of LoRaWANs through lightweight scheduling. *IEEE Internet of Things Journal*, 5(3):1830–1842, 2018.

[27] Abusayeed Saifullah, Mahbubur Rahman, Dali Ismail, Chenyang Lu, Ranveer Chandra, and Jie Liu. SNOW: Sensor network over white spaces. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*, pages 272–285, 2016.

[28] Abusayeed Saifullah, Mahbubur Rahman, Dali Ismail, Chenyang Lu, Jie Liu, and Ranveer Chandra. Low-power wide-area network over white spaces. *IEEE/ACM Transactions on Networking*, 26(4):1893–1906, 2018.

[29] Mariusz Slabicki, Gopika Premsankar, and Mario Di Francesco. Adaptive configuration of LoRa networks for dense IoT deployments. In *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*, pages 1–9. IEEE, 2018.

[30] STMicroelectronics. STM32L151RE, 2020. https://www.st.com/en/microcontrollers-microprocessors/stm32l151re.html.

[31] RED Technologies. Wave DB Connect forTVWS, 2021. https://www.redtechnologies.fr/sas-technology-copy.

[32] Deepak Vasisht, Zerina Kapetanovic, Jongho Won, Xinxin Jin, Ranveer Chandra, Sudipta Sinha, Ashish Kapoor, Madhusudhan Sudarshan, and Sean Stratman. FarmBeats: An IoT platform for data-driven agriculture. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 515–529, 2017.

A Compliance of Emission

We conduct laboratory measurements to ensure that the emission generated by Whisper radio complies with the FCC leakage regulations. Our evaluation is twofold. First, we look at the changes in the desired to undesired (D/U) signal ratio on the first adjacent channel of select DTV receivers when the source of the undesired signal is changed from a broadband white space device (WSD) to a NB WSD (Whisper radio). Second, we evaluate the changes in the D/U ratio on the first adjacent channel of select DTV receivers in response to the airtime of Whisper radio.

A.1 Setup

As the bandwidth of a NB TVWS channel is limited to 100 kHz, we select the lowest and highest possible bandwidths supported by the Whisper radio within the limit. Recall that airtime in LoRa modulation is determined by the spreading factor (SF). In this case, we use the lowest (SF7) and highest (SF12) possible spreading factors to change the airtime from low to high, respectively. The duty cycle of the Whisper IoT radio was set at 78% (ON time 780 ms, OFF time 220 ms for each second transmission). Note that the FCC proposed duty cycle is considerably less: 1%.

As shown in Table 3, a set of DTVR - RX1, RX4, RX5, RX10, and RX12 - are identified for this evaluation to diversify over different price ranges, resolutions, dimensions, and form factors. Channel 9 in the high-VHF band (center frequency: 189 MHz) and Channel 16 in the UHF band (center frequency: 485 MHz) are selected to provide the desired DTV signal. The D/U ratio for the Whisper radio is average across four desired signal levels – moderately strong (-43 dBm), moderate (-53 dBm), moderately weak (-65 dBm), and very weak (-80 dBm) at ± 3 MHz and ± 6 MHz from the edge of the broadcast DTV channel. These are the same desired signal levels used in the broadband WSD laboratory testing. In this way, the two sets of measurements for the D/U ratio on the first adjacent channel can be compared. The video loop used in the D/U measurements for the broadband WSD measurements is used for this test. For each measurement, the undesired signal power is increased until artifacts are observed.

A.2 Results

Table 4 and 5 summarize the results of D/U ratio evaluation for different DTVR receivers.

For the ATSC 1.0 DTVs, (RX1, RX4, RX5, and RX10), the D/U ratio on the first adjacent channel for the NB TVWS IoT radio indicates the receivers are even more selective (i.e., the value of the D/U ratio more negative) with respect to an undesired NB WSD than an undesired broadband WSD. Note that the D/U ratio at ± 6 MHz from the broadcast channel's edge is usually a few dB better (more selective) than the D/U ratio at ± 3 MHz from the broadcast channel's edge.

For the ATSC 3.0 receiver (RX12) tests at a modulation level of 256QAM, the D/U ratio for NB and broadband WSDs for moderate and weak desired signals are about the same, within error. There is a 5-6 dB difference for moderately strong and very weak desired signal, where the NB WSD is more selective than the broadband WSD for moderately strong signals and the NB WSD is less selective than the broadband WSD for very weak signals. In both instances, the D/U ratio represents very high ATSC 3.0 receiver selectivity. As RX12 is tested only at 256 QAM, it could be a function of the higher threshold at this modulation level.

DTVR ID	Manufacturer	Model	Standard	Profile	Resolution	Display size
RX1	Samsung	UN65NU8000	ATSC1.0	TV	4K	65in
RX4	Samsung	UN32N5300AFXZA	ATSC1.0	TV	1080p	32in
RX5	Insignia	NS-24DF310NA19	ATSC1.0	TV	720p	32in
RX10	Mediasonic	HOMEWORX HW130STB	ATSC1.0	Set-up box	1080p	-
RX12	RedZone	TVXPLORER BUNDLE	ATSC3.0	USB dongle	HD	-

Table 3: Information of DTVR receivers

DTV center frequency = 189MHz (Channel 9)			
	Change in D/U when SF is increased from SF7 to SF12		Change in D/U when b/w is increased from 7.8kHz to 62.5 kHz
DTV Receiver	7.8kHz	62.5kHz	SF7	SF12
RX1	-0.1	1.4	1.8	0.3
RX4	0.9	1.4	0.8	0.3
RX5	0.1	1.2	1.5	0.4
RX10	0.1	8.1	8.0	0.0
RX12	0.5	2.5	3.1	0.2

Table 4: Change in the D/U Ratio of DTV Receivers for the DTV Transmitter Operating on Channel 9

DTV center frequency = 485MHz (Channel 16)			
	Change in D/U when SF is increased from SF7 to SF12		Change in D/U when b/w is increased from 7.8kHz to 62.5 kHz
DTV Receiver	7.8kHz	62.5kHz	SF7	SF12
RX1	-0.2	1.7	1.9	0.0
RX4	0.6	1.0	0.6	0.2
RX5	0.1	0.7	1.0	0.2
RX10	0.1	4.7	5.0	0.2
RX12	0.3	1.1	1.3	0.0

Table 5: Change in the D/U Ratio of DTV Receivers for the DTV Transmitter Operating on Channel 16

In general, the NB WSD operating at 62.5 kHz bandwidth with a spreading factor of SF12 displays higher impact on DTVR operation compared to other configurations. For the traditional ATSC 1.0 DTV receivers (RX1, RX4, and RX5), there is negligible change in the D/U ratio for the NB WSD operating at bandwidths of 7.8 kHz (at spreading factors SF7 and SF12) and 62.5 kHz (at spreading factors SF7 and SF12), for measurements in both the UHF and high-VHF bands. For the low-cost digital-to-digital converter (RX10), there is significant change in the D/U ratio with the 62.5 kHz bandwidth and SF12, with a greater change observed in the high-VHF band than the UHF band. For the ATSC 3.0 receiver, there was a $2-3$ dB change in the D/U ratio observed in the high-VHF frequency band for the 62.5 kHz bandwidth and SF12. The change in the UHF frequency band was minimal: $1.1 - 1.3$ dB.

B   MAC Protocol

B.1   Handling collision in quantum selection

If two assignments – A_1 and A_2 – of a scheduling quantum are colliding, we can make these collision free by ensuring that one of these keeps silence in the colliding hour. For example, if A_2 keeps silence, then we represent collision free

form of it as $A_2 < h_{s2}, p_2, Silent[<h_{s1}, p_1>]>$. Inside $Silent[]$, we can keep all the assignments colliding with A_2. In this way, the new assignment is made collision free with respect to the existing assignments of a quantum. Now, the question arises, how to measure the effect of silencing? Here, we introduce a new metric, frequency of silence (ω_q), representing how frequently the new assignment requires to be silent. We can get the value of ω_q from the generic form of solution to Diophantine equation. In the aforementioned example, ω_q for A_2 is quotient of p_1 divided by $gcd(p_1, p_2)$. If $\omega_q = 1$, then the new assignment requires to be silent in every occurrence of it to avoid collision. Consequently, the quantum is of no use for the new assignment. Based on it, if the quantum has existing assignments (mutually non-colliding by definition) that collide with the new one, we use $\omega_{q\%} = \frac{number\, of\, assignments\, impelling\, \omega_q\, frequency\, of\, silence}{\omega}$ for quantum selection. If $\omega_{q\%} \geq 1$, then the new assignment requires to be silent in every occurrence of it. Note that the value of $\omega_{q\%}$ is clamped at 1 for any further calculation.

B.2   Construction of channel occupancy table

In Figure 4, A_{c0} is the first assignment for the channel with the occupancy of $\tau = 6$. The first table in the figure represents the entry of A_{c0}. The first entry in the table with no assignment is to facilitate future entries. A_{c1} is the next assignment with the occupancy of $\tau = 9$ and does not collide with A_{c0}. Now, there are 2^2 possible combination of these two assignments as shown in the second table which is basically a binary counter of two digits. However, A_{c0} and A_{c1} do not collide, and thus the last entry in the second table has $\tau = 0$. Consequently, any combination in future having these two assignments will not have a common colliding hour. So we can remove this entry and get the third table. A_{c2} is the next assignment with the occupancy of $\tau = 12$ and collides with both A_{c0} and A_{c1}. Using the method of binary counting we derive the fourth table from the third one. Finally, if the value of τ crosses the limit of $36sec$ in an entry while adding a new assignment, the assignment is not valid for the channel.

B.3   Pseudo code of slot allocation algorithm

In Algorithm 1, we show how possible combinations of slots are prepared. We first get the dictionary of collision free scheduling quanta for all base station radios using scheduling quantum selection as mentioned above. The output of

Algorithm 1: Pseudo code: Generate possible combination of slots for slot allocation algorithm

```
Function Get_PossibleSlotCombinations(p, σ, αmin):
    GET qDictionary FROM Quantum_Selection(p);
    INIT possibleSlotCmbntns[] TO empty;
    while qDictionary != empty do
        POP entry<A, collisionFreeq[]> FROM
          qDictionary;
        for α = αmax TO αmin do
            if collisionFreeq != empty then
                INIT listOfGrpdq[] TO (SELECT * FROM
                  (SELECT * FROM collisionFreeq GROUPBY
                  (continuity = α)) ORDERBY (AVG(ρq));
                INIT cntGrpdq TO Count(listOfGrpdq);
                INIT numOfSlotRqrd TO ⌈σ/α⌉;
                INIT cntSlotGotChnl TO zero;
                INIT slotsWithChnl[] TO empty;
                for i = 0 TO cntGrpdq − 1 do
                    INIT Ac TO <A, τ, listOfGrpdq[i]>;
                    GET channel FROM Channel_Selection (Ac);
                    if channel != null then
                        ADD <listOfGrpdq[i], channel> TO
                          slotsWithChnl[];
                        INCREASE cntSlotGotChnl BY 1;
                        if cntSlotGotChnl == numOfSlotRqrd then
                            ADD <slotsWithChnl[], α, Avg(ρq), ρc, A>
                              TO possibleSlotCmbntns[];
                            break;
```

scheduling quantum selection is a dictionary having different possible (with(/out) modification for silencing) assignments ($A<h_s, p, Silent[]>$) as the key and corresponding list of collision free scheduling quanta as the value. For each entry in the dictionary, we then vary the continuity (α) of the scheduling quantum from max to given min. For each continuity value, we group the α consecutive scheduling quanta. Groups are ordered by the average ρ_q of member scheduling quanta in each group. Here, to serve the requirement of σ scheduling quanta, we take $\lceil \frac{\sigma}{\alpha} \rceil$ groups of scheduling quantum. Next, we select downlink channel for each group using the channel selection algorithm. Now, each group is an individual slot, and the combination of $\lceil \frac{\sigma}{\alpha} \rceil$ slots together satisfy the requirement from the client. In this way, we make a list of the possible combinations of slots. Next, to select a combination from the list, we use the above mentioned weight function.

B.4   Cases where *p* is out of boundary

Now, what if the traffic has a periodicity of p_{low} seconds that is less than 1hr? We convert it as $\lceil \frac{p_{low}}{3600} \rceil$ requirements hav-

ing a periodicity of 1hr. During the slot allocation, we select slot combinations for each requirement having a time gap of no more than p_{low} seconds from the selected slot combinations for earlier requirement. If a client generates periodic traffic with $p > 24hr$, it freshly joins the network before data transmission. Because as mentioned in Section 5.4, a static client with periodic traffic requires to poll channel status from WSDB once in every 24 hours with the recent GPS reading. The poll request is piggybacked as a MAC command in a confirmed up data frame. As a client generating traffic at a periodicity of more than 24hr, it does not send any data frame within 24hr of the pervious one where the poll request can be piggybacked. Besides, sending a frame just for channel polling would be power inefficient. Thus, it it freshly joins the network every time when it has data frame to transmit. It in turn saves the power of polling channel in every 24 hours.

B.5   Client bootstrapping

B.5.1   Beacon

In one hour, there are N_{bcnprd} pre-specified beacon periods, each having $N_{bcnslot}$ of the same length. Every beacon slot has identical periodicity of p_{bcn}. For example, in Figure 5, each beacon period has $N_{bcnslot} = 2$ beacon slots with a periodicity of $p_{bcn} = 1hr$. A NB channel for downlink transmission is associated with each beacon slot. These NB beacon channels are distinctly picked from the TV channels registered for the gateway which increases robustness against noisy link and interference. The beacon slot structure along with channels are pre-specified and pre-loaded in the client radio. Here, the question arises what would happen if the WSDB in future ceases transmission on a particular beacon channel for the gateway? Although it is very infrequent, it needs to be addressed to comply with the regulation. In such a case, the gateway goes silent in the beacon slot associated with ceased channel. However, the clients continue to listen on that channel without violating the regulation. Finally, N_{bcnprd}, $N_{bcnslot}$, and p_{bcn} are adjusted based on application and expected traffic pattern. For example, for a deployment expecting high event-driven traffic, N_{bcnprd} is set to a higher value with lower p_{bcn}.

B.5.2   Join

On reboot, a client radio first locks the GPS and retrieves the current location and time with a precision of one second. It then hops across the beacon channels according to the previously specified beacon slot structure and listens for the beacon. Since the client is already time synced using GPS and knows beacon schedule, it requires minimal hopping depending on the channel quality. Each beacon frame embeds join slot information as MAC command in an encrypted LoRaWAN multicast frame. To be specific, each beacon frame contains the scheduling info of N_{jnslot} join slots between the

current and next beacon period. Hence, the info received in a beacon is only valid till the next beacon period. For example, in Figure 5, each beacon contains the scheduling info of following $N_{jnslot} = 3$ join slots. Here, the gateway books the join slots as an event-driven traffic. The benefit is twofold. First, N_{jnslot} can be adjusted depending on the expected join requests in a deployment over the time. Second, it utilizes no or low-utilized channels in that hour assigned for periodic assignments. It in turn impels variation in the channels used in the join slots with the time which increases robustness against noisy channels and long-term interference. Note that the uplink and downlink channels are same in the join slot.

Upon receiving the beacon, a client selects one of the join slots from the info embedded in the beacon. Since multiple clients may attempt to send join request at the same time, we need to ensure that clients are selecting join slots in a distributed manner to reduce the collision. Here, the client leverages dynamic quadratic hash function where the unique device id is used as the key. The gateway decides the coefficients of the hash function equation. To do so, it estimates the set of clients likely to send join request using set the difference between the provisioned clients for the deployment and clients already joined the network for periodic traffic. The coefficients chosen based on this estimated set are embedded in the beacon. For example, in Figure 5, $Client_0$ and $Client_1$ select two different join slots.

In the join slot, a client transmits the join request along with its location and slot requirement for the data communication. Upon receiving the join request, the gateway sends the client's location to the channel coordinator for the registration on the WSDB (see Figure 2). Once the registration of the client is done, the Whisper MAC manager gets the available channel for it and allocates the slots for data communication. The gateway then sends a join response incorporating the allocated slots and expiry of the associated channels.

Learning to Communicate Effectively Between Battery-free Devices

Kai Geissdoerfer
TU Dresden

Marco Zimmerling
TU Dresden

Abstract

Successful wireless communication requires that sender and receiver are operational at the same time. This requirement is difficult to satisfy in battery-free networks, where the energy harvested from ambient sources varies across time and space and is often too weak to continuously power the devices. We present Bonito, the first connection protocol for battery-free systems that enables reliable and efficient bi-directional communication between intermittently powered nodes. We collect and analyze real-world energy-harvesting traces from five diverse scenarios involving solar panels and piezoelectric harvesters, and find that the nodes' charging times approximately follow well-known distributions. Bonito learns a model of these distributions online and adapts the nodes' wake-up times so that sender and receiver are operational at the same time, enabling successful communication. Experiments with battery-free prototype nodes built from off-the-shelf hardware components demonstrate that our design improves the average throughput by 10–80× compared with the state of the art.

1 Introduction

The last few years have seen rapid innovation in battery-free systems [40], culminating in a number of real-world applications [1, 12, 27]. These systems pave the way toward a more sustainable Internet of Things (IoT) [7] by enabling small, cheap, and lightweight devices to perform complex tasks (e.g., DNN inference [20]) off ambient energy while using tiny, environmentally friendly capacitors as energy storage [40]. However, to replace today's trillions of battery-powered IoT nodes, battery-free devices must learn to communicate.

Challenge. The power that can be harvested from solar, vibrations, or radio signals is typically insufficient to continuously operate a device. A traditional energy-neutral device buffers harvested energy in a rechargeable battery and can *freely control* its average duty cycle to avoid power failures. Instead, a battery-free device cannot avoid power failures, and has *very limited control* over when the power failures begin and end. Fig. 1 illustrates this so-called *intermittent operation*. After executing for a short time, a battery-free device is *forced* to
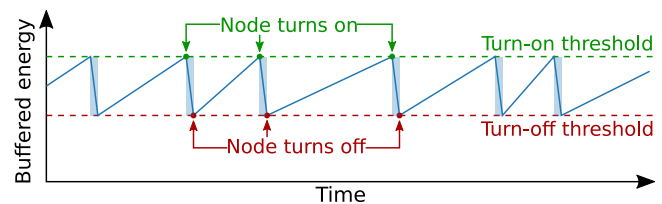
Figure 1: Because ambient power is often weak, a battery-free node must buffer energy before it can wake up and operate for a short time period. This is known as intermittent operation.

become inactive and wait for a long, fluctuating time until its capacitor is sufficiently charged again. For example, when harvesting energy from indoor light, our prototype battery-free nodes need to stay off and recharge, on average, for hundreds of milliseconds before they can operate for at most 1 ms.

Many techniques have been developed to deal with intermittency on a *single* battery-free device [6, 11, 34], but how to communicate *between* intermittently powered devices is one of the most pressing problems yet to be solved [22, 28, 48]. This is due to the fact that device-to-device communication is a fundamental building block for a variety of network and system services, including optimal clock synchronization [26], ranging and localization [9, 21], sensor calibration [41], distribution and coordination of sensing and computing tasks [32], collaborative learning [47], and efficient and reliable wireless networking [25]. Realizing these services across battery-free devices has the potential to enable novel and more sustainable IoT and sensor network applications, from automatic contact tracing to planetary-scale environmental monitoring.

To be able to communicate, sender and receiver must be active and have enough energy for at least one complete packet transmission *at the same time*. However, since the nodes' activity phases are generally interleaved and short compared to their charging times, as visible from the real-world trace in Fig. 2a, it often takes thousands of wake-ups until two nodes encounter each other and communication becomes possible [19]. Moreover, after an encounter, the nodes quickly get out of sync if they become active immediately after a
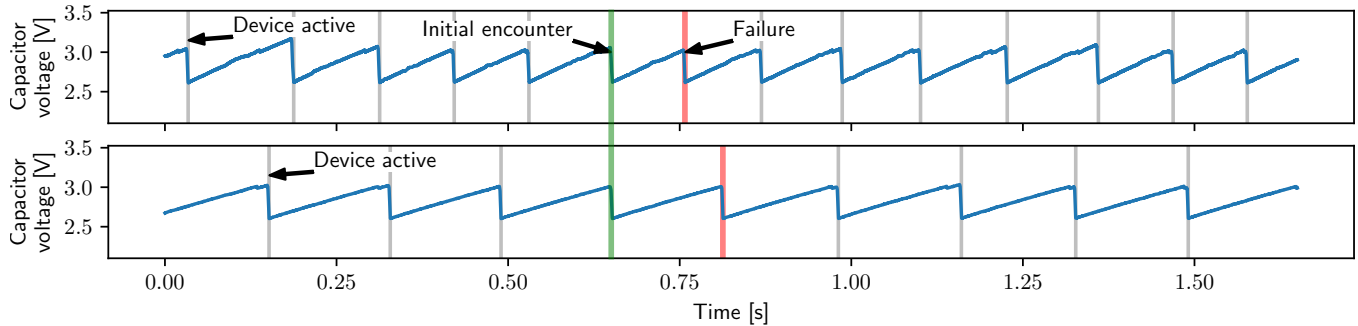
(a) Because of their short and interleaved activity phases, battery-free devices often need a long time with hundreds of wake-ups until they encounter each other. Even after an initial encounter, the devices quickly get out of sync, rendering communication inefficient and unreliable.
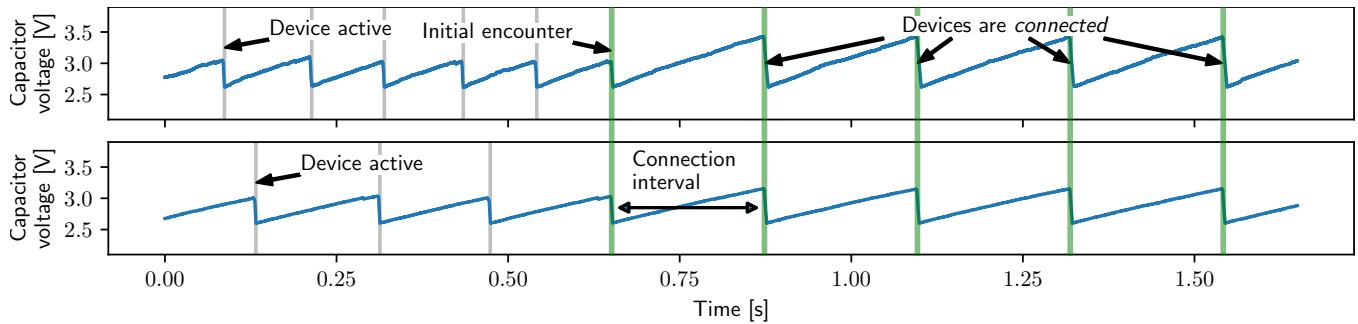
(b) With Bonito, devices learn and exchange statistical models of their charging times and agree on a connection interval that ensures that both devices have sufficient energy at the same time. Maintaining a connection over multiple encounters enables efficient and timely communication.

Figure 2: The challenge of efficient battery-free device-to-device communication in (a) and our proposed protocol in (b).

recharge, as stipulated by the state of the art [8, 33] and apparent in Fig. 2a. This is because ambient energy varies across time and space [3], which leads to fluctuating and different charging times between the nodes.

Besides establishing a first encounter [19], active radio communication has been considered too demanding for battery-free devices [36]. Conversely, work on backscatter communication has focused on physical-layer issues, such as improving range and throughput, purposely considering high-energy environments, batteries, or cables to continuously power the devices in the experiments to avoid intermittency [29, 35, 38, 49]. However, when running off ambient energy, duty cycling of the backscatter transceivers becomes necessary [14, 29, 43]—and, without a battery, the intermittency problem occurs.

Contribution. This paper presents Bonito, the first connection protocol for battery-free wireless networks. Bonito provides reliable and efficient bi-directional communication despite the time-varying intermittency of battery-free devices.

The real-world trace in Fig. 2b illustrates the high-level protocol operation. Unlike the state of the art, Bonito enables two battery-free nodes, after an initial encounter, to maintain a *connection* across multiple consecutive encounters. To this end, Bonito continually adapts the *connection interval*, which is the time between the end of an encounter and the beginning of the next encounter. A shorter connection interval provides more communication opportunities in the long run. However, a connection interval that is shorter than any of the nodes' charging times breaks the connection and requires the nodes to wait for a long time until they encounter each other again. Thus, the challenge is to keep the connection interval as short as possible without losing the connection, which is difficult in the face of time-varying charging times.

One of our key insights is that, depending on the scenario and energy-harvesting modality, the charging time of a battery-free node approximately follows well-known probability distributions. We leverage this insight in Bonito by letting each node *continuously learn and track* the parameters of a model that approximates the distribution of locally observed charging times against non-stationary effects (e.g., changes in mean or variance). Then, to maintain an efficient and reliable connection, the nodes exchange at every encounter their current model parameters and jointly adapt the connection interval.

We implement Bonito on a custom-designed ultra low-power battery-free node. Our prototype is built from off-the-shelf components, including an ARM Cortex-M4 microcontroller that features a 2.4 GHz Bluetooth Low Energy (BLE) radio. The node harvests energy from a solar panel or a piezoelectric harvester, using a 47 µF capacitor as energy storage.

To evaluate Bonito through testbed experiments and fairly compare it against two baselines, we use up to 6 Shepherd ob-
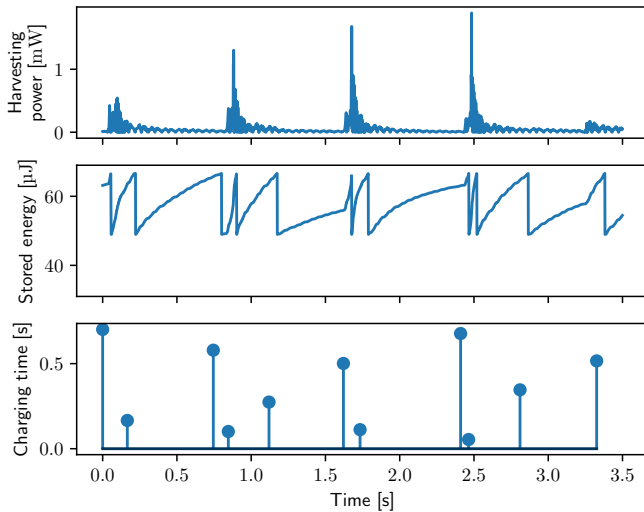
Figure 3: The top plot shows an example trace of real kinetic harvesting power during jogging (see picture in Fig. 4a). The middle and bottom plots show the corresponding energy stored in the capacitor and the resulting charging times of a simulated battery-free device.

(a) Runner with full measurement setup for the *jogging* dataset.

(b) Washing machine with partial setup for the *washer* dataset.

Figure 4: Pictures from two of the five scenarios in which we use synchronized Shepherd nodes [18] to record energy-harvesting traces.

servers [18] to record and replay real-world energy-harvesting traces from 5 diverse scenarios. Our results show, for example, that Bonito maintains connections for hundreds of consecutive encounters, and that it outperforms the state of the art by 10–80× in terms of throughput. We also conduct a case study that demonstrates the utility of Bonito for accurate and timely occupancy monitoring in homes and commercial buildings.

Overall, this paper contributes the following:

- We collect 32 h of energy-harvesting traces from 5 different scenarios. Our analysis of these traces provides new insights into spatio-temporal intermittency patterns.
- We design the Bonito protocol. Bonito enables, for the first time, reliable and efficient communication between intermittently powered battery-free devices.
- We demonstrate an efficient implementation of Bonito on a prototype node with a 3.1 mm³ ceramic capacitor.
- Results from testbed experiments and an occupancy monitoring case study provide evidence that Bonito performs well under a diverse range of real-world conditions.

2 Motivation

While previous work on intermittency has focused on individual battery-free devices [6,11,33] or discovery of neighboring devices [19], reliable and efficient device-to-device communication is still an open challenge. By *device-to-device communication* we mean the regular exchange of application data between two battery-free devices after they have successfully discovered each other through a first encounter [19].

To motivate the need for our work, we consider the scenario of battery-free wearables. Fig. 3 shows real-world data from a piezoelectric energy harvester that is attached to the ankle of a person (see Fig. 4a). The upper plot shows the harvest-

ing power while the person is jogging, recorded by a Shepherd node. Shepherd is a measurement tool that records time-synchronized voltage and current traces from one or more energy-harvesting nodes with high rate and resolution [18]. The power spikes correspond to when the foot strikes the ground, with significantly lower harvesting power during the rest of the stride cycle. Based on trace-driven simulations, the middle plot shows the corresponding amount of harvested energy stored in an ideal 17 µF capacitor powering a battery-free device that turns on when the capacitor voltage exceeds 3 V and turns off when the capacitor voltage falls below 2 V. We see that when the device powers up, the stored energy is quickly consumed, forcing it to turn off already after about 1 ms. While powered off the harvesting power exceeds the standby power, so energy is accumulated and the capacitor voltage rises again. Compared to the short activity phases, the time needed to charge the capacitor, shown in the bottom plot of Fig. 3, is much longer and varies significantly over time.
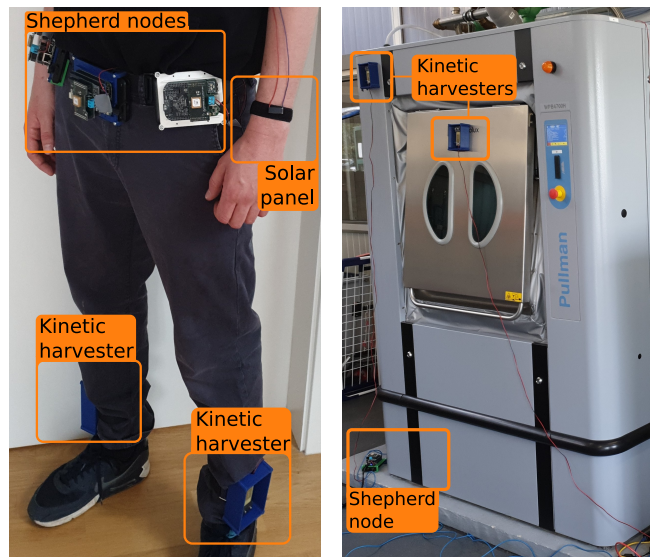
The variability of a node's charging time is a function of its location and the associated energy environment, that is, how much power the harvester delivers at any given time. Thus, two battery-free devices, even when they are physically close to each other, have a different energy environment and therefore experience different charging times.

As an example, Fig. 5 plots the charging times of two devices during jogging over one hour. One device is powered by a piezoelectric harvester attached to the left ankle of a person, while the other device is powered by the same type of harvester attached to the right ankle of the person (see Fig. 4a). Each point in Fig. 5 indicates the charging times of both de-

Dataset	Energy Source	Harvester Part Number	Duration	#Devices	#Links	#Wake-ups	Model
Jogging	Human motion	MIDE S128-J1FR-1808YB	1 h	3	10	13252	Exponential
	Outdoor solar	IXYS KXOB25-05X3F		2		119127	Normal
Stairs	Outdoor solar	IXYS KXOB25-05X3F	1 h	6	15	359002	Normal
Office	Indoor light	IXYS SM141K06L	1 h	5	10	98324	Gaussian mixture
Cars	Car vibrations	MIDE S128-J1FR-1808YB	2 h	6	15	8517	Exponential
Washer	Machine vibrations	MIDE S128-J1FR-1808YB	45 min	5	10	22224	Normal

Table 1: Overview of energy-harvesting datasets we record in a variety of scenarios.
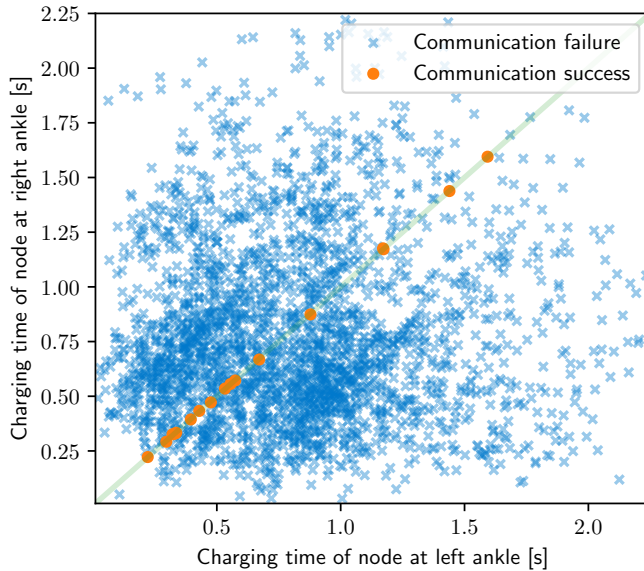
Figure 5: Charging times of two battery-free devices powered by kinetic harvesters attached to a jogger's ankles (see Fig. 4a). Using the greedy approach, the devices communicate successfully only in 0.04 % of the cases in which the charging times are almost identical.

Figure 6: Success rate of greedy approach in trace-driven simulations, averaged across all pairs of devices (i.e., links) in a given dataset.

vices when they begin to charge their $17\,\mu F$ capacitors at the same time from the same initial charge. We observe that in many instances the two nodes have vastly different charging times. This means that if nodes become active as soon as they reach the turn-on threshold, which is the state-of-the-art approach, called *greedy* and illustrated in Fig. 2a, the nodes often wake up with an offset that prevents communication, despite a successful encounter at the previous wake-up. Indeed, the success rate for the two nodes in Fig. 5 is less than 0.04 %. This leads to poor communication reliability and efficiency as the nodes more often than not fail to exchange their data.

To assess the generality of these observations, we record distributed energy-harvesting traces in diverse scenarios using multiple Shepherd nodes [18]. Table 1 lists the main characteristics of the five datasets we collected:

- The full *jogging* dataset comprises traces from two participants, each equipped with two piezoelectric harvester at the ankles and a solar panel at the left wrist (see Fig. 4a). The two participants run together for an hour in a public

park, including short walking and standing breaks.
- For the *stairs* dataset, we recorded traces from six solar panels that are embedded into the surface of an outdoor stair in front of a lecture hall. Over the course of one hour, numerous students pass the stairs, leading to temporary shadowing effects on some or all of the solar panels.
- The *office* dataset comprises traces from five solar panels mounted on the doorframe and walls of an office with fluorescent lights. During the one-hour recording, people enter and leave the office and operate the lights.
- The *cars* dataset contains traces from two cars. Each car is equipped with three piezoelectric harvesters mounted on the windshield, the dashboard, and in the trunk. The cars drive for two hours in convoy over a variety of roads.
- The *washer* dataset includes five traces from piezoelectric harvesters mounted on a WPB4700H industrial washing machine, as shown in Fig. 4b, while the machine runs a washing program with maximum load for 45 min.

Fig. 6 plots for each dataset the average success rate across all pairs of devices (i.e., communication links) in the scenario. Even in the most favorable scenario, *stairs*, where the solar panels receive a fairly constant and similar energy input from natural sunlight, we find that the greedy approach succeeds in only 16 % of the cases. In all other scenarios, the success rate ranges below 3.5 %. Our experiments on real battery-free nodes in Sec. 5 confirm these trace-driven simulation results.

3 The Bonito Protocol

This section describes the Bonito protocol. The Bonito protocol enables two battery-free devices to stay connected after a first encounter, which can happen either coincidentally or with the support of a neighbor discovery protocol [19].

3.1 Overview

Bonito aims to make nodes repeatedly encounter each other so they can exchange application data reliably and efficiently, as shown in Fig. 2b. To ensure that nodes wake-up with a time offset small enough for a successful encounter, they agree at every encounter on a new *connection interval* T_C. This is the time between the end of the current encounter and the beginning of the next (i.e., planned) encounter.

Main idea and approach. For two nodes i and j with known charging times c_i and c_j, the shortest possible connection interval T_C^* is simply the maximum of their charging times

$$T_C^* = max(c_i, c_j) \tag{1}$$

If a shorter connection interval $T_C < T_C^*$ is used, then one node does not reach the required energy level to become active by T_C. Thus, the encounter fails, preventing the nodes from agreeing on the next connection interval—the connection is *lost*. A lost connection entails that the nodes often need to wait for a long time until they encounter each other again to resume communication. However, choosing a longer connection interval $T_C > T_C^*$ to mitigate the risk of a lost connection adds unnecessary delay as nodes, after having reached the required energy level, are forced to wait before they wake up at T_C.

The key challenge is to determine the connection interval T_C such that both nodes have enough energy while introducing only minimal delay. This is difficult as the charging times c_i and c_j are unknown and time-varying, as discussed in Sec. 2.

Using a probabilistic approach, we address this problem as follows. Let p be the probability that nodes i and j have sufficient energy to become active after a connection interval T_C. This corresponds to the probability that the nodes' charging times, c_i and c_j, are shorter than the connection interval T_C. Modeling c_i and c_j as random variables with a strictly monotonically increasing joint cumulative distribution function (cdf) $F_{i,j}$, this translates into

$$p = F_{i,j}(c_i = T_C, c_j = T_C) \tag{2}$$

Solving for T_C yields the minimum connection interval that guarantees, with a user-defined probability p, a successful encounter of the two nodes at their next wake-up

$$T_C = F_{i,j}^{-1}(p) \tag{3}$$

where $F_{i,j}^{-1}$ is the inverse joint cdf of c_i and c_j.

Base protocol. In practice, the joint cdf $F_{i,j}$ is rarely known a priori. Moreover, $F_{i,j}$ can only be estimated online by the nodes based on full knowledge of each other's charging times.

Unfortunately, this requires frequent communication between battery-free nodes—precisely what Bonito intends to enable.

To circumvent this chicken-and-egg problem, we assume that the charging times, c_i and c_j, are statistically independent. In this case, the joint cdf $F_{i,j}$ is the product of the marginal cdfs F_i and F_j. The marginal cdfs can be estimated locally by each node from observations of their own charging times.

Based on these insights, we propose the following main steps of the Bonito protocol:

1. Each node i continuously estimates the marginal cdf F_i of its charging time based on local measurements.
2. When two nodes i and j encounter each other, they exchange their current estimates of F_i and F_j.
3. Using the same inputs (i.e., the marginal cdfs F_i and F_j and the user-defined probability p), both nodes compute the same new connection interval T_C according to (3).
4. Both nodes become active and communicate after the new connection interval T_C, and continue with step 2.

In this way, Bonito adapts the connection interval to changes in the energy environment, effectively enabling battery-free nodes to stay connected across several hundreds of subsequent encounters, as demonstrated by our experiments in Sec. 5.

To achieve this performance, we first need to answer the following key questions in our design of Bonito:

- How to compactly represent and exchange the marginal cdfs F_i and F_j in the face of limited energy (Sec. 3.2)?
- How to learn and track online an accurate estimate of F_i against a changing energy environment? (Sec. 3.3)
- How to efficiently compute the inverse joint cdf $F_{i,j}^{-1}(p)$ to obtain the connection interval T_C? (Sec. 3.4)

3.2 Modeling Charging Time Distributions

Because of the small energy storage, battery-free devices can only exchange a limited amount of data during an encounter. Thus, the marginal cdfs F_i and F_j must be represented in a compact form in order to be able to exchange them.

Unlike the common belief that the duration of a recharge is completely random [11, 31], we make the empirical observation that, in the scenarios we considered, the nodes' charging times can be faithfully modeled by well-known distributions. The rightmost column of Table 1 lists the models we use for each dataset. To illustrate, Fig. 7 plots representative charging time distributions and the corresponding models for the stairs, cars, and office datasets. Non-stationary effects like a time-varying mean are removed in the plots as these are effectively handled by our online learning approach detailed in Sec. 3.3.

We observe in Fig. 7a that when harvesting energy from outdoor solar with a constant harvesting voltage, the charging time can be modeled by a normally distributed random variable. The intuition is that temporary environmental effects, such as shadowing and change in incidence angle, let the charging time vary around a certain value. Fig. 7b shows that an exponential distribution is often a good fit when harvesting kinetic energy. This can be explained by the decaying
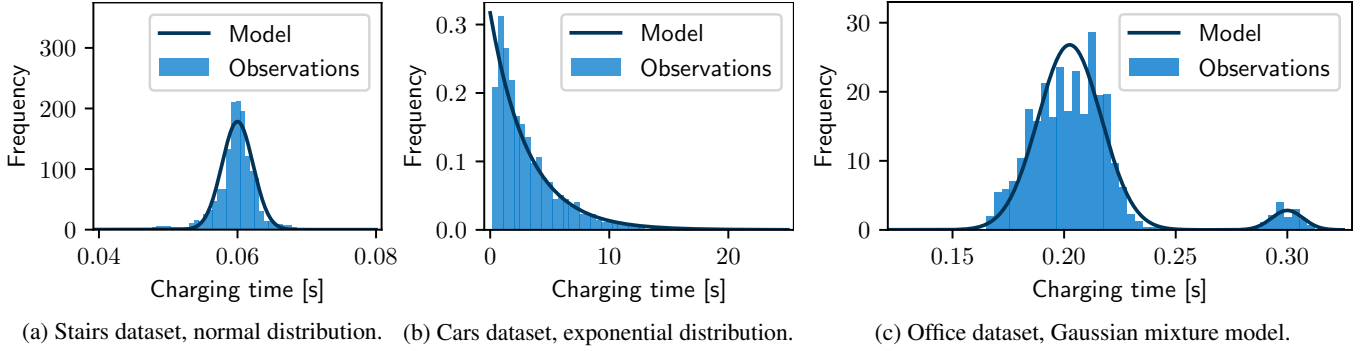
(a) Stairs dataset, normal distribution.   (b) Cars dataset, exponential distribution.   (c) Office dataset, Gaussian mixture model.

Figure 7: Charging time distributions of individual nodes. The nodes' charging times can be modeled by well-known distributions.

response of a piezoelectric harvester to the distinct impulses of a car during driving (e.g., acceleration, breaking, bumps) or a person during jogging (see Fig. 3). In the washer scenario, instead, we find that the continuous shaking of the industrial washing machine over long periods induces approximately normally distributed charging times. Looking at Fig. 7c, we see that in the office scenario the charging times are mostly distributed around a certain value. However, the maximum power point tracking (MPPT) of the DC-DC converter used in this scenario, which periodically disconnects the charger for a short time, leads to a second peak. We approximate this distribution with a Gaussian mixture model (GMM).

These observations motivate us to model the marginal cdf F_i of a node's charging time in the scenarios we considered through the parameters of a normal distribution (2 parameters), an exponential distribution (1 parameter), or a GMM (6 parameters for two Gaussians and two weights). The last column of Table 1 lists the corresponding model for each of the datasets. The jogging dataset contains traces from different types of harvesters: We use an exponential distribution to model the charging times of kinetic harvesting nodes and a normal distribution for the solar harvesting nodes. During an encounter, a node only needs to share the type of model and the current estimates of the model parameters.

3.3 Learning Distribution Parameters Online

We now turn to the problem of estimating the parameters of a given charging time distribution based on local observations. Given a sample of n independent and identically distributed observations, the log-likelihood $\mathcal{L}(\theta \mid x)$ and the corresponding maximum likelihood estimator $\widehat{\theta}$ are given by

$$\mathcal{L}(\theta \mid x) = \ln\left(\prod_{i=1}^{n} f_\theta(x_i)\right) = \sum_{i=1}^{n} \ln f_\theta(x_i) \qquad (4)$$

$$\widehat{\theta} = \arg\max_{\theta} \mathcal{L}(\theta \mid x) \qquad (5)$$

where $f_\theta(x_i)$ is the conditional probability to observe x_i if the underlying distribution is parameterized with θ.

Unfortunately, vanilla maximum likelihood estimation is not viable in our setting. First, the observations of the charg-
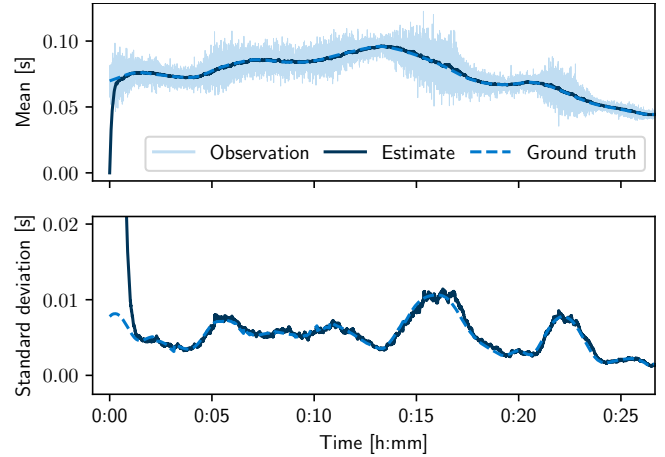
Figure 8: Varying mean and standard deviation over a moving window of one of the trace from the stairs dataset reveal non-stationarity. Using SGD, the changing distribution parameters are tracked online.

ing time become available only one by one at runtime, yet the nodes do not have enough memory and energy to recompute the estimator with every new observation. Further, the charging time distributions are non-stationary. For instance, the dashed lines in Fig. 8 reveal trends in the mean and standard deviation of a node's charging time from the stairs dataset. Thus, an approach is needed that dynamically adjusts the parameter estimates to changing energy harvesting conditions.

To address these problems, Bonito learns the distribution parameters online using stochastic gradient descent (SGD), which has become a popular method for training a wide range of machine learning models [5]. Compared to a sliding window based approach, SGD is less computationally demanding as the parameter update is only computed for the current observation rather than for a set of past observations that also have to be kept in memory. If the gradient of $\mathcal{L}(\theta \mid x)$ is known, one solution to (5) is to iteratively adjust $\widehat{\theta}$ along the gradient, known as gradient descent

$$\nabla \mathcal{L}(\theta \mid x) = \nabla \sum_{i=1}^{n} \ln f_\theta(x_i) \qquad (6)$$

$$\widehat{\theta} = \widehat{\theta} + \eta \cdot \nabla \mathcal{L}(\widehat{\theta} \mid x) \qquad (7)$$

By pulling the ∇ operator in (6) into the sum, the update step in (7) can be split into a series of updates for every individual observation x_i. This yields the update equation of SGD

$$\widehat{\theta}_i = \widehat{\theta}_{i-1} + \eta \cdot \nabla \mathcal{L}(\widehat{\theta} \mid x_i) \qquad (8)$$

Sec. A derives the gradient equations required to solve (8) for the normal, exponential, and Gaussian mixture models. By keeping the learning rate η constant, Bonito implicitly reduces the weight of old observations relative to more recent observations. This way, devices dynamically learn changing properties of the charging time distribution locally, without information exchange with other devices.

Example. Fig. 8 illustrates how Bonito learns and tracks mean and standard deviation of a non-stationary normal distribution. To obtain ground truth, we sample charging times (i.e., observations) from a known normal distribution, whose mean and variance change dynamically over time. We extract these changes from one of the traces in the stairs dataset using a 2 min moving average filter. We can see in Fig. 8 that the parameter estimates of Bonito converge from their initial values (zero mean and unit standard deviation) to the true ground truth parameters within less than a minute. Then the estimates closely follow the changes of the underlying distribution.

3.4 Computing Inverse Joint CDF Efficiently

Having shared the type of model and the current estimates of the model parameters during an encounter, Bonito needs to compute the new connection interval T_C from the inverse joint cdf $F_{i,j}^{-1}$ for a user-defined probability p. This is difficult since there exists no closed-form solution for most bivariate distributions, let alone for joint cdfs of different distribution families (e.g., when a solar and a kinetic energy harvesting node in the jogging scenario want to communicate). Instead, we have to solve (3) numerically, while taking into account the energy and compute constraints of battery-free devices.

We are interested in the connection interval T_C where the joint cdf is equal to the user-defined target probability, that is, $F_{i,j}(T_C) = p$. This yields the following objective function

$$f(T_C) = F_{i,j}(T_C) - p = F_i(T_C) \cdot F_j(T_C) - p = 0 \qquad (9)$$

Note that $f(T_C)$ has a single root—the sought solution—as $F_{i,j}$ is strictly monotonically increasing. Bonito solves this problem using the well-known bisection method, which iteratively finds the root of any continuous function that has its root inside a bracket (i.e., search interval). Indeed, we can derive such a bracket based on the inverse cdfs of our marginal distributions, which either have a closed form solution (exponential and normal) or are easy to approximate (GMM).

To derive a lower bracket, we first note that $F(x) < 1$ for any cdf F. It follows that $F_{i,j}(x=z, y=z) = F_i(x=z) \cdot F_j(y=z) < \min(F_i(x=z), F_j(y=z))$ and therefore the lower bracket

$$F_{i,j}^{-1}(p) > \max\left(F_i^{-1}(p), F_j^{-1}(p) \right) \qquad (10)$$
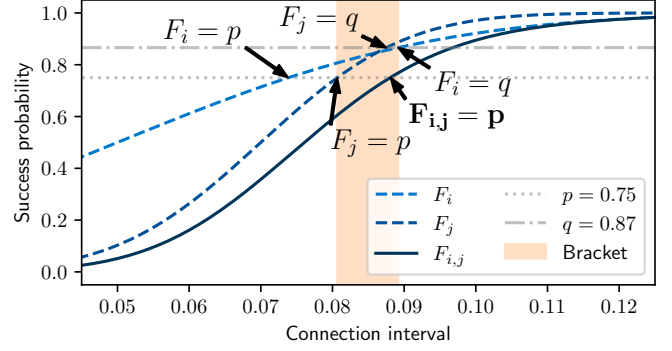
Figure 9: Bracketing the inverse joint cdf based on the inverse cdf of the marginal distributions enables efficient computation of the connection interval on resource-constrained battery-free devices.

To derive an upper bracket, we introduce $q = \sqrt{p}$ and $c = \max(F_i^{-1}(q), F_j^{-1}(q))$. Let F_m be the marginal cdf (i.e., either F_i or F_j) such that $F_m^{-1}(q) = c$, that is, the marginal cdf that reaches q *later*. Let F_n be the other marginal cdf that reaches q *sooner*. From $F_n(c) \geq F_m(c)$ follows $F_{i,j}(c) = F_m(c) \cdot F_n(c) \geq F_m(c) \cdot F_m(c) = q^2 = p$. Finally, because $F_{i,j}(c)$ is monotonically increasing, we obtain the upper bracket

$$F_{i,j}^{-1}(p) \leq \max(F_i^{-1}(q), F_j^{-1}(q)) \qquad (11)$$

Example. Fig. 9 shows an example, where (10) and (11) are used to determine an initial bracket for $F_{i,j}^{-1}(p = 0.75)$. The resulting bracket $[0.61, 0.77]$ is already relatively tight, and therefore we find the solution $F_{i,j}^{-1}(p = 0.75) = 0.88$ with a tolerance of 0.01 after only three bisection steps.

3.5 Impact of Target Probability

The target probability p is a key parameter of the Bonito protocol that must be set by the user. It specifies the probability that both devices have accumulated enough energy in their capacitors to become active after a connection interval T_C. A high target probability requires a long connection interval T_C, increasing communication delay and lowering throughput.

To illustrate how the choice of p impacts communication reliability and efficiency, we run trace-driven simulations as detailed in Sec. 2 on the traces from the datasets in Table 1. We use two metrics to quantify the performance of Bonito: As a proxy for communication reliability, we define the *success rate* as the ratio of successful encounters with Bonito to the total number of wake-ups. As a proxy for communication efficiency, we consider the *relative delay* as the median of all successful connection intervals with Bonito divided by the median of the optimal clairvoyant solutions according to (1).

Fig. 10 plots for each dataset success rate and relative delay averaged across all links. We can observe the following:

- A higher target probability p leads to a higher success rate, which demonstrates the plausibility of our approach.
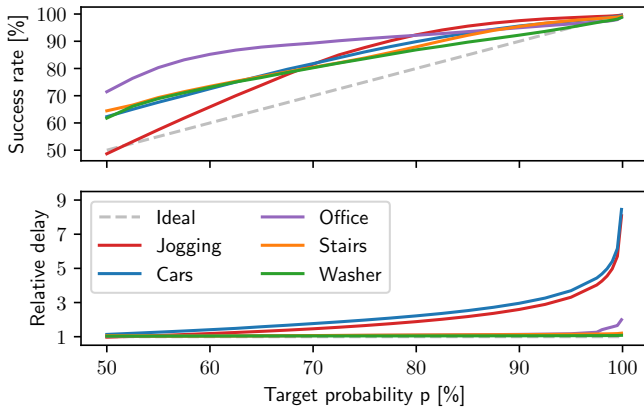
Figure 10: Trace-driven simulations reveal that the rate of successfully arranged encounters matches the user-defined target probability. The price to pay in terms of latency depends on the model of the underlying charging time distribution.

In most cases, the success rate is even slightly higher than requested, presumably due to small model errors.

- Since connection losses are costly, a higher target probability is preferable in practice. Fig. 10 shows that the price to pay in terms of a higher relative delay depends on the scenario. For the cars and jogging datasets, where most or all links include at least one node with approximately exponentially distributed charging times, the relative delay increases exponentially with p, due to the heavy tail of the distribution. For GMM (office), the increase is moderate, whereas it is hardly noticeable for the normal distribution (washer and stairs).

4 Implementation

In this section, we describe the hardware and software components of our prototype implementation.

4.1 Hardware

We design a ulta low-power battery-free node based on the popular Nordic Semiconductor nRF52805 microcontroller (MCU). This particular MCU features a 2.4 GHz BLE radio and a state-of-the-art 32-bit 64 MHz ARM Cortex-M4, which is powerful enough to complete also more demanding computations in a short time, benefitting overall system efficiency. To enable low-power timekeeping between wake-ups, the MCU is equipped with a 32 kHz crystal with ± 20 ppm frequency tolerance. A TI BQ25504 DC-DC step-up converter charges a 2 mm × 1.25 mm × 1.25 mm 47 µF multilayer ceramic capacitor (MLCC) from a connected solar panel or a piezoelectric energy harvester. Once the capacitor voltage reaches a hardware-programmable threshold of 3.3 V, the BQ25504 sets one of its pins high. This pin is wired to a TI TS5A23166 analog switch that connects the MCU to the capacitor-buffered supply voltage.

Due to its DC bias characteristics, the capacitor has an ef-
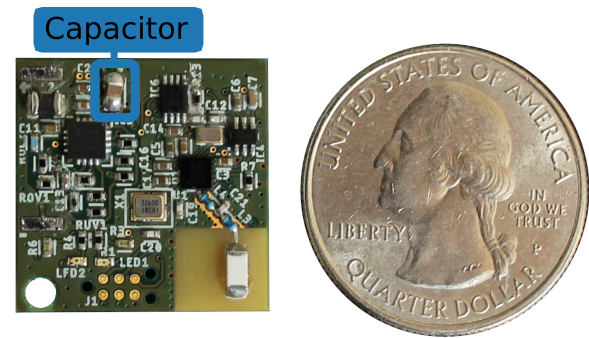
Figure 11: Prototype battery-free node based on the nRF52805 MCU. A sustainable 3.1 mm^3 ceramic capacitor is used as energy storage.
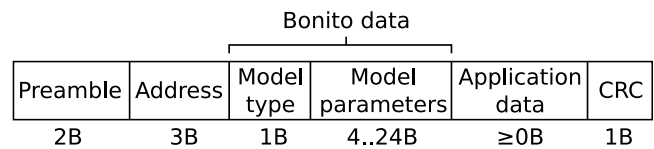
Figure 12: Packet format. Using Bonito, nodes exchange between 5 B and 25 B carrying model type and parameters during an encounter.

fective capacitance of only 17 µF at 3.3 V. This allows for a maximum active time of around 1 ms per wake-up. A larger capacitance would increase the active time per wake-up and the charging time between wake-ups. To minimize the physical dimensions and the price of the node, we choose the minimum capacitance that allows the nodes to remain active for long enough to compensate for clock drift accumulated over a connection interval of 5 s (see Sec. 4.2).

The node also integrates a circuit to measure the current flow from the harvester, which can be used as a sensing signal [39]. The two-layer printed circuit board (PCB) shown in Fig. 11 measures 20 mm × 20 mm. The total cost of all components is $8.73.

4.2 Software

We implement Bonito and the Find neighbor discovery protocol [19] on our battery-free nodes. Find is used to establish an initial encounter after a connection loss or a power failure.

Bonito protocol settings. We use the 2 Mbit/s BLE mode and the frame structure depicted in Fig. 12. Depending on the model type, encoded by one byte, a packet carries 1, 2, or 6 model parameters represented by 32-bit floating point values.

To jointly agree on the next connection interval, Bonito requires nodes to exchange messages bi-directionally during an encounter. The exact sequence of packet exchanges is subject to application requirements and can be flexibly configured. We implement the packet sequence shown in Fig. 13. When two nodes encounter each other using Find, one of the nodes receives the first beacon and replies with an acknowledgement. At all following encounters, the node that received the first beacon starts to listen at the time agreed on using Bonito.
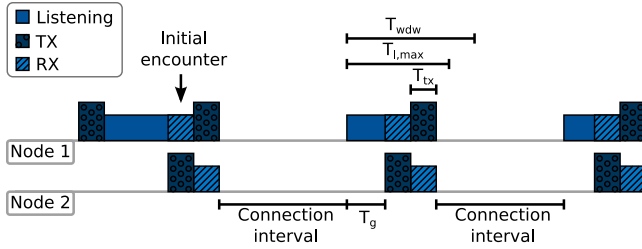
Figure 13: After the initial encounter, nodes use Bonito to agree on a connection interval. At the next encounter, one of the nodes starts to listen and the other node transmits its packet after a grace period to account for clock drift due to the long charging times. After receiving this packet, the listening nodes replies with its own packet.

Due to the small energy buffer, a node can keep the radio on for at most $T_{wdw} = 1$ ms. Thus, the maximum listening time is $T_{l,max} = T_{wdw} - T_{tx} - T_{ta} = 820\,\mu s$, where $T_{ta} \approx 40\,\mu s$ is the time it takes to switch from receive to transmit mode and $T_{tx} = 140\,\mu s$ is the airtime of a packet with 6 model parameters and 4 B of application data. To increase the robustness to clock drift in the face of long charging times and hence long connection intervals, we let the node that sends first transmit its packet after a grace period of $T_g = 0.5 \cdot T_{l,max} - 1.5 \cdot T_{tx} = 200\,\mu s$. We can thus tolerate an offset of up to $\pm 200\,\mu s$ between the clocks of the two nodes, which corresponds to a maximum connection interval of 5 s when taking into account the frequency tolerance of the 32 kHz crystal oscillator. Upon receiving the packet, the other node switches to transmit mode and sends its own packet.

In our current implementation of Bonito, the devices consider a connection as lost whenever a planned packet exchange fails, for example, due to fading, external interference, or when one of the two devices does not reach the turn-on threshold by the end of the connection interval. In this case, they return to discovery mode and use Find to re-establish the connection.

Runtime support. In addition to Bonito and Find, we implement an efficient *soft intermittency* runtime, where the MCU is gracefully suspended to an ultra low-power mode before an impeding power failure [19]. This reduces the costs associated with a cold start after a hardware reset and allows to keep track of time between consecutive wake-up events using the built-in real-time clock (RTC). To this end, a node periodically samples the capacitor voltage during charging with the built-in analog-to-digital converter (ADC) until the capacitor voltage reaches a software-defined turn-on threshold. Then the node executes protocol and application code until it is interrupted by the power-fail comparator, upon which it immediately transitions back into low-power mode to replenish its energy buffer.

Although our runtime tries to prevent hardware resets, after multiple seconds without any energy input, the sleep current drains the remaining charge from the capacitor and the node eventually powers off. While powered off, the on-board static random access memory (SRAM) is subject to decay, that is, bits that were set to one may flip and become zero after some time. To still retain the trained model of a node's charging time distribution across short power failures, we store it in a dedicated section of the SRAM. After every model update, we compute a checksum over this section and store it next to the model parameters. If the recomputed checksum after a hardware reset does not match the checksum stored in memory, we conclude that the memory is corrupted and restart training the model with the initial parameters.

5 Evaluation

This section uses testbed experiments to evaluate Bonito on real battery-free nodes under realistic, repeatable conditions. We start by showing in Sec. 5.2 how Bonito dynamically adjusts the connection interval to changes in the nodes' charging times to maintain long-running connections. In Sec. 5.3, we compare Bonito against two baseline approaches. Finally, in Sec. 5.4, we quantify the runtime overhead of Bonito. Our experiments reveal the following key findings:

- Bonito establishes connections that outlast on average hundreds of consecutive encounters even between nodes that harvest from different types of energy sources.
- Bonito improves the throughput by 10–80× compared with the current state of the art. It achieves this by consciously keeping the connection interval as short as possible while maintaining a high success rate that agrees to within 1 % of the requested target probability.
- Depending on the distribution model, Bonito consumes between 4 % and 25 % of the energy available per wake-up on our nodes. The energy cost of losing a connection is 1000× higher than the energy overhead of Bonito.

5.1 Testbed and Settings

We connect two battery-free nodes (see Fig. 11) to two Shepherd observers [18]. In addition to recording spatio-temporal harvesting traces (see Sec. 2), Shepherd can also replay previously recorded traces and monitor the behavior of connected battery-free devices. The observers synchronously replay for all 60 links in our datasets (see Table 1) the two corresponding energy-harvesting traces. At the same time, the observers log the serial output and GPIO events of the attached nodes, which we use to compute performance metrics. In total, we collect measurements from 218 hours of testbed experiments.

For the stairs, office, and washer scenarios, we replay the recorded energy-harvesting traces as is. When using the original traces from the cars and jogging scenarios, however, we were not able to collect sufficient data points. The reason is that the piezoelectric harvesters were selected and tuned for the frequency and amplitude of the washer scenario, which led to a relatively low harvesting power in the cars and jogging scenarios, as evident from the small number of wake-ups in Table 1. Because it can take thousands of wake-ups until two nodes encounter each other, we had to scale the cars and jogging traces by a factor of five to allow for a meaningful
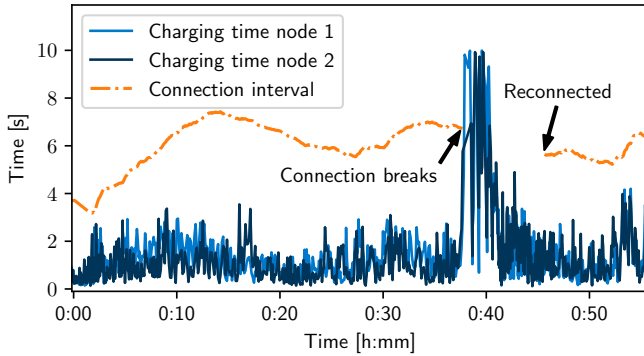
Figure 14: Real-world trace from testbed experiments showing the charging times of two nodes from the cars dataset. By dynamically adjusting the connection interval, Bonito maintains a connection for 37 min until the cars leave the highway and enter stop-and-go traffic; the charging times increase dramatically and the connection breaks.

evaluation. Note that this does not change the dynamics and shape of the charging time distributions, nor does it affect relative performance when comparing different approaches.

In all experiments, we configure Bonito with a target probability of $p = 0.99$. We use a learning rate of $\eta = 0.01$ for the normal and exponential models and $\eta = 0.001$ for GMM, which we found to perform well in a wide range of scenarios.

5.2 Maintaining Long-running Connections

We begin by looking at how well Bonito can maintain a connection between battery-free devices. As an illustrative example, Fig. 14 shows the charging times of two nodes from the cars dataset and the connection interval determined by Bonito over the course of 55 min. Bonito successfully maintains the connection for more than half an hour by dynamically adjusting the connection interval based on the continuously updated models of the nodes' charging time distributions. Then, after around 37 min, the two cars driving in convoy exit the highway and enter stop-and-go traffic. As a result, the charging times increase suddenly and exceed the connection interval—the connection is lost. At this point, the nodes switch over to executing the Find neighbor discovery protocol and successfully reconnect after roughly 10 min. Afterward, Bonito takes over and again maintains a connection for several minutes.

Fig. 15a plots for all datasets the cdf of the connection duration in terms of the number of encounters, while Fig. 15b plots it in terms of time for the unscaled datasets (see Sec. 5.1). Overall, we find that in 90 % of the cases, the nodes stay connected for at least 30 consecutive encounters, and 40 % of the connections last for 800 encounters or more. This demonstrates that Bonito enables, for the first time, reliable and efficient communication between intermittently powered nodes.

5.3 Bonito versus Baseline Approaches

We now compare Bonito against two baseline approaches:

- Greedy: This is the current state of the art. Using Greedy, nodes wake up and attempt to communicate as soon as
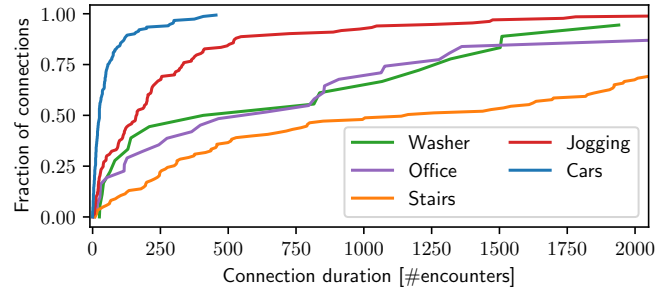
(a) cdf of connection duration in terms of number of encounters.
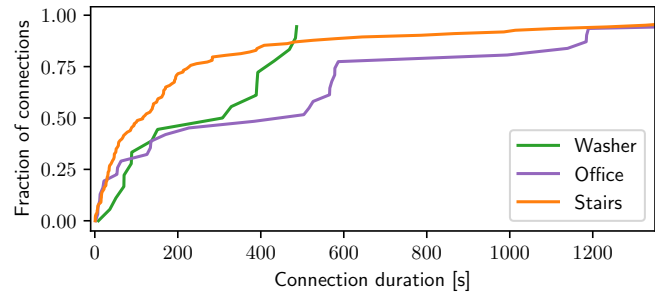
(b) cdf of connection duration in terms of time.

Figure 15: Bonito maintains connections over hundreds of encounters even in challenging scenarios with different types of energy sources.

they reach the minimum required energy level. Greedy is the prevalent execution model in the intermittent computing literature [8,33] as it maximizes the effective duty cycle of a battery-free device.

- Modest: As a complementary approach to Greedy, we design Modest. Using Modest, each node keeps track of the maximum observed charging time c_{max}. During an encounter, two nodes i and j share their current maximum charging times $c_{max,i}$ and $c_{max,j}$, and agree to meet again after a connection interval of $T_C = \max(c_{max,i}, c_{max,j})$.

Our comparison uses two end-to-end metrics that also account for periods where Find runs to establish a first encounter after a connection loss or power failure. *Throughput* is the number of packets delivered from one node to another node per time unit. Note that traffic is always bi-directional, that is, the same number of packets is also delivered in the other direction (see Fig. 13). *Latency* is the time between two consecutive packet exchanges. We also consider *success rate*, which is the ratio of successfully arranged encounters to the total number of trials when using Greedy, Modest, or Bonito.

Fig. 16 plots for each dataset the throughput gains of Bonito and Modest over Greedy. We see that Bonito improves the throughput by 10–80×. For example, for the stairs dataset, Bonito achieves a throughput of 15.18 pkt/s versus 0.33 pkt/s with Greedy. Modest outperforms Greedy across the board, too, but often falls far short of Bonito's throughput.

To understand the reasons for the significant performance differences among the different approaches, we plot in Fig. 17 success rate and latency for the stairs dataset. As the charging
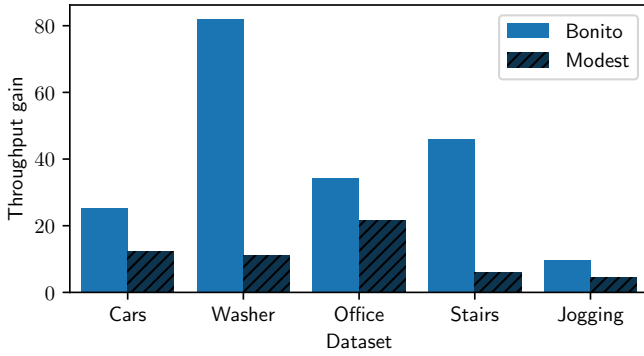
Figure 16: Throughput improvement over Greedy. By maintaining connections over many wake-ups, the average number of encounters with Bonito is at least an order of magnitude higher than with Greedy.
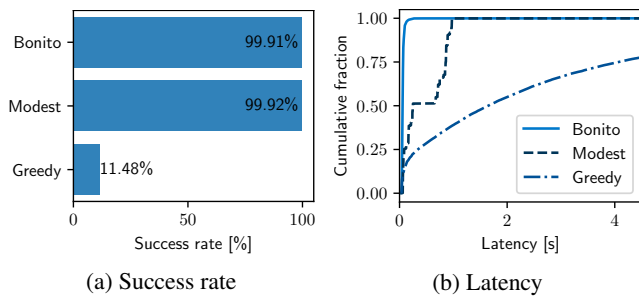

(a) Success rate   (b) Latency

Figure 17: Detailed comparison of performance metrics for the stairs scenario. Bonito achieves a high success rate that is on a par with the Modest approach, while providing a significantly lower latency.

times vary across time and space, Greedy achieves a low success rate of only 11.48 % (see Fig. 17a). This means that in 9 out of 10 cases the nodes loses the connection right after the first encounter. Every time the connection is lost, the nodes cannot communicate until they reconnect, causing excessively long latencies as visible in Fig. 17b. Instead, Modest chooses the connection interval highly conservatively, which leads to a high success rate of 99.92 % but also long latencies. Bonito provides much shorter latencies at almost the same high success rate, which agrees to within 1 % of the requested target probability. By aiming to keep the connection interval short and to avoid the latency associated with reconnecting after a connection loss, Bonito significantly increases the end-to-end throughput compared with the two baseline approaches.

5.4 Bonito's Runtime Overhead

Next, we evaluate the runtime overhead of Bonito based on the logs from the testbed experiments. The overhead can be broken down into three components: (*i*) updating the model parameters using SGD, (*ii*) exchange of the model parameters over wireless during an encounter, and (*iii*) computing the inverse joint cdf to obtain the connection interval.

The time required to update the model is constant: 1.3 μs for exponential, 3.2 μs for normal, and 28.8 μs for GMM. This constitutes up to 2.8 % of the around 1 ms active time per wake-up. Similarly, the airtime to exchange 4, 8, or 24 bytes
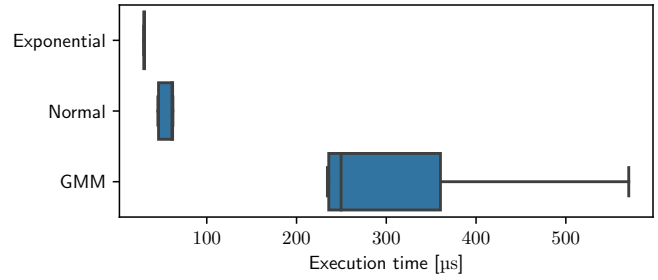
Figure 18: Distribution of execution times on our battery-free node when computing the inverse joint cdf. The execution time depends on the number of model parameters and varies with the number of bisection steps needed to satisfy the required tolerance.
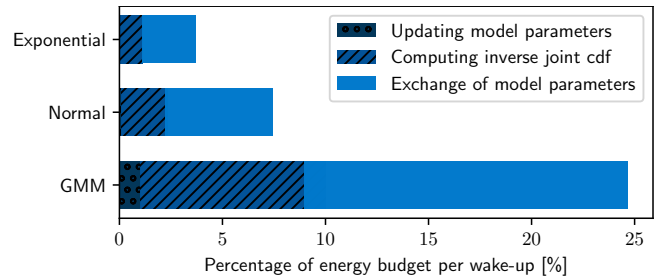
Figure 19: The energy overhead of Bonito ranges between 4 % and 25 % of the energy available per wake-up on our nodes. In absolute terms, the cost to recover from a lost connection is 1000× higher.

of model parameters is fixed and determined by the bitrate of the BLE radio. By contrast, Fig. 18 shows that the time to compute the inverse joint cdf varies depending on the number of bisection steps required to reach the desired tolerance.

In terms of energy, our battery-free nodes have an energy budget of 27.5 μJ per wake-up. Fig. 19 shows for each model the median percentage of energy budget spent by Bonito. We can see that the required energy mainly depends on the number of model parameters and the computational complexity of evaluating the inverse joint cdf. In the worst case, for GMM, Bonito consumes 7.1 μJ, which amounts to about 25 % of the available energy per wake-up. To set this into perspective, Fig. 20a plots for all datasets the time it takes for two nodes to synchronize with the Find neighbor discovery protocol [19] in terms of the number of wake-ups, while Fig. 20b plots it in terms of time for the unscaled datasets (see Sec. 5.1). On average it takes 283 wake-ups, or 7782.5 μJ, to synchronize after a lost connection—1000× more than the energy required by Bonito to maintain a connection. This demonstrates that, overall, the absolute energy costs of Bonito are well spent.

6 Case Study: Occupancy Monitoring

Occupancy monitoring is essential to save energy in homes and commercial buildings [10, 16]. Recently, it has also become an important tool to manage the spread of infectious diseases, such as SARS-CoV2 [37]. To assess the potential of Bonito for real-world battery-free applications, we conduct an
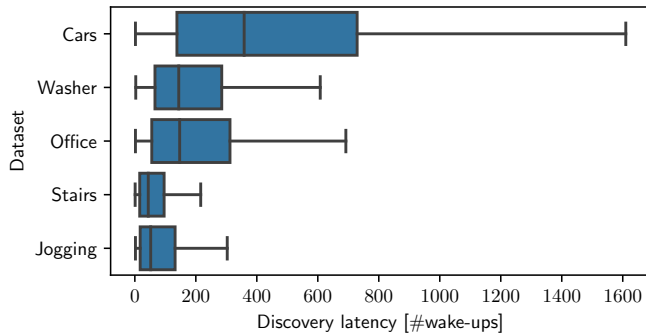
(a) Discovery latency in terms of number of wake-ups.
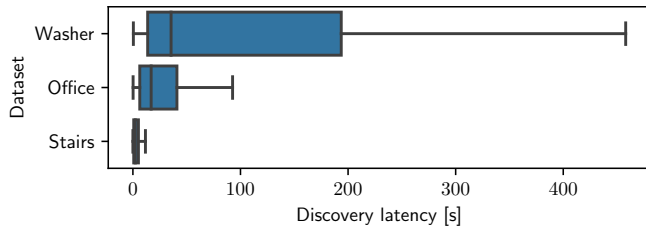

(b) Discovery latency in terms of time.

Figure 20: Synchronizing two devices with the Find neighbor discovery protocol takes a long time and consumes significant energy. Using Bonito, devices can establish long-running connections to periodically exchange data without the need to resynchronize.

occupancy monitoring case study with our prototype nodes.

Occupancy sensor. To efficiently count the number of people in a room, we use the solar panel as a sensor [23, 39] to detect when a person enters or leaves the room. Fig. 21 shows the solar panel current of two nodes mounted next to each other on a doorframe (see Fig. 22), when a person enters the room in Fig. 21a and when a person leaves the room in Fig. 21b. To detect the direction of movement, the nodes record the time when they detect the onset of the shadowing by the person. Then the nodes exchange the recorded times and compute the time difference τ. The sign of τ indicates the direction.

Setup. We mount two battery-free nodes equipped with IXYS SM141K06L solar panels next to each other on the doorframe at the entrance of an office room, as shown in Fig. 22. The nodes sample the solar panel current with a sampling rate of 1 kHz, and record the time when the solar panel current falls below 87.5 % of its average value. The nodes run Bonito and insert the timestamp of detected events into the packets. Together with logging information (charging time, connection interval, etc.) every packet carries 26 B of application data.

Because the clocks of the two nodes are not synchronized, timestamps are transmitted relative to the start of the corresponding packet. To this end, nodes measure the time between the detected event and the start of the transmission and insert the result into the packet. The receiving node timestamps the reception of the packet and converts the contained relative timestamp to its local clock. Finally, by relating a received
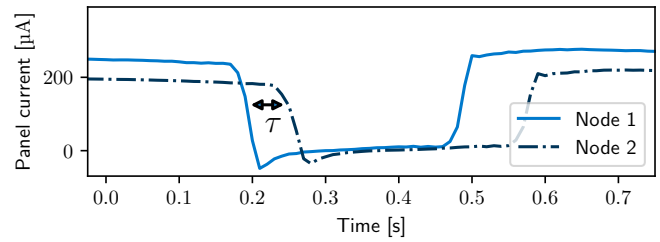
(a) Person entering the room: $\tau > 0$
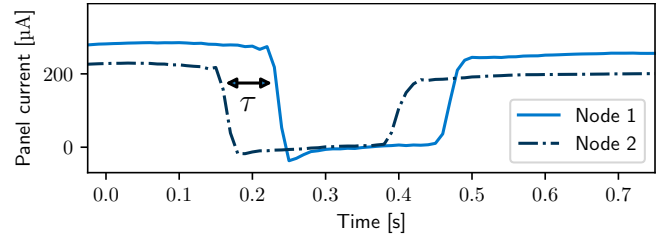
(b) Person leaving the room: $\tau < 0$

Figure 21: The shadow of a person passing ambient light harvesting devices on a doorframe causes a distinct temporal pattern in the solar panel current. By comparing the times of the onset of the shadowing on the two nodes, we can determine the direction of movement.

timestamp to the timestamp of the corresponding event that was recorded locally, the nodes compute the time difference τ.

The nodes transmit the result over wireless to an nRF52840 development board that serves as a base station. We configure the base station to timestamp the reception of packets containing a detected event and button presses of two on-board push buttons, one for each direction. Four participants randomly enter and leave the room one by one. Another person records ground truth by pressing the corresponding button on the nRF52840 board precisely when a person passes through the doorframe.

Results. The confusion matrix in Table 2 shows that the system correctly classified 60 out of 61 events, corresponding to an accuracy of 96 %. It missed just one in-event, and falsely reported an in-event and an out-event for a single in-event.

Fig. 23 plots the latency in terms of the time between a
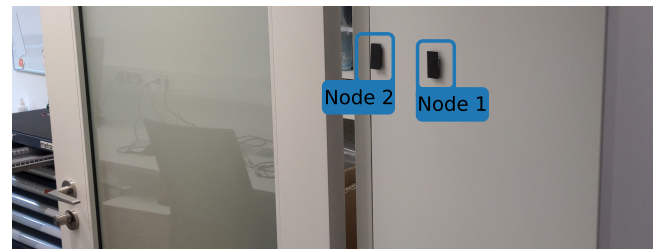

Figure 22: Two of our battery-free nodes are attached to the doorframe and harvest energy from ambient light. Thanks to Bonito, the nodes can communicate in a timely and reliable fashion, allowing them to count the number of people entering and leaving the room.

		Ground truth		
		In	Out	No event
	In	30	0	1
Recorded	Out	0	31	0
	No event	1	0	0

Table 2: By collaborating, the battery-free nodes classified people entering and leaving the room with an average accuracy of 96.83 %.
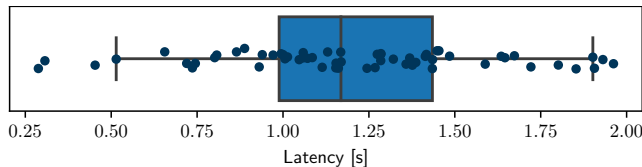

Figure 23: Due to the low communication latency provided by Bonito, the system reported detected events both timely and accurately.
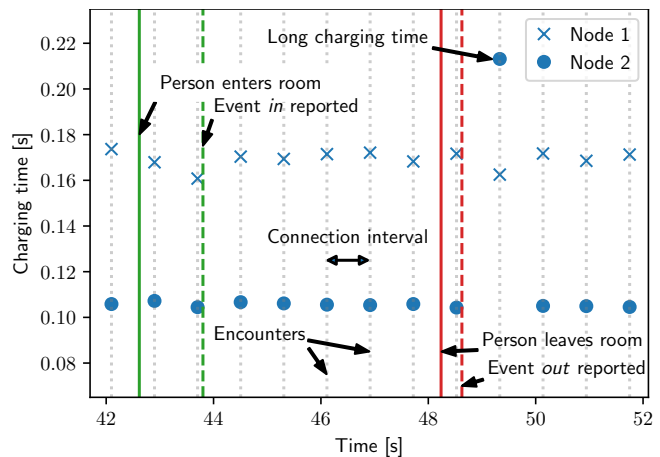

Figure 24: Example trace from the occupancy monitoring case study. The system correctly classifies and reports events to the base station. With Bonito, the connection interval is chosen large enough to sustain outliers of the charging time in response to transient shadowing.

button press and the reception of the detected event at the base station. The median latency was 1.2 s and all events were reported within less than 2 s. Over the course of the experiment, the two nodes successfully exchanged 10.56 kB of application data for an application-level throughput of 28.38 B/s.

Fig. 24 shows a ten-second excerpt from the experiment. The markers indicate the charging times of the nodes. Solid vertical lines indicate button presses (ground truth); dashed vertical lines indicate when a event was received at the base station. We can observe that, right after the received out-event, node 1 reports an exceptionally high charging time of 210 ms. This happens when the shadowing by a person occurs while a node charges its capacitor: The shadowing reduces the energy input for a short time, which prolongs the recharge. Nevertheless, by keeping the connection interval at around 700 ms, Bonito provides a stable connection despite such dynamics.

7 Discussion

Bonito is the first connection protocol for battery-free devices. It enables two devices to communicate efficiently and reliably by dynamically adapting the connection interval to changes in the devices' energy availability. In this section, we discuss limitations and opportunities for extending Bonito.

From connections to networks. The ability to efficiently and reliably exchange data between two devices is the fundamental building block required to form large wireless networks consisting of multiple battery-free devices. A number of tradeoffs and challenges arise from each of the possible approaches to move from the two-node setting to larger networks, which could be explored by future work. For example, devices may sequentially connect with their neighbors or devices may try to establish Bonito connections with one common connection interval between multiple devices.

Communication with battery-powered devices. While we focus on communication between two battery-free devices, Bonito is also useful for effective communication from battery-

free to battery-powered devices. For example, a battery-free tag may want to transmit data to a user's smartphone or to a battery-powered gateway in a wireless sensor network. Because battery-powered devices are in control of their wake-up times, any connection interval works for them. Thus, instead of computing the inverse joint cdf of the charging time distribution of both devices, it is sufficient to compute the inverse cdf of the charging time of the battery-free device in order to determine a connection interval that works for both devices.

Model accuracy. The goodness-of-fit of the learned charging time model critically affects the performance of Bonito. With perfect knowledge of the underlying distribution, Bonito would compute the minimum connection interval feasible for the requested target probability. Overestimating the real distribution leads to increased delay, while underestimation reduces reliability. If the distribution is so complex that a large number of model parameters or a non-parametric model (e.g., a deep neural network) would be required to accurately capture this complexity, then the limited resources on a battery-free device may not be sufficient to learn the model online.

Exploiting statistical dependence. In the current implementation, Bonito assumes statistical independence of the nodes' charging time distributions in order to compute a connection interval without prior knowledge of the statistical properties of the joint charging time distribution. After establishing a connection, the devices can record observations of the joint distribution and could attempt to exploit statistical dependence between their charging times, possibly improving communication efficiency and reliability.

8 Related Work

Intermittent computing. The thriving research area of intermittent computing has made great strides in recent years, including the first real deployments of battery-free sensors [1].

This achievement rests upon techniques that ensure forward progress [34], consistent peripheral state [6], and a reliable notion of time [11] despite frequent and random power failures. This line of research is highly relevant but completely orthogonal to our work as it deals exclusively with intermittency issues on *individual* devices and, if at all, considers communication with continuously powered base stations [42].

Battery-free device-to-device communication. Prior work on battery-free wireless device-to-device communication is mainly theoretical [24, 30, 50], studying the energy trade-offs for different scheduling, transmission, and decoding policies. Recent work discusses middleware and applications for networks of intermittently powered devices, yet explicitly leaves the question of how to communicate between the devices as an open problem [28, 48]. A simulative study also acknowledges the sheer difficulty of sychronizing the wake-up times of intermittently powered devices and proposes to communicate an energy state via an always-on backscatter radio, without demonstrating a real implementation or experiments [45]. Similar to Bonito, a recent theoretical work proposes to let nodes agree on a future point in time when they become active to increase communication throughput [46]. This time is computed based on a moving average of previous charging times, whereas Bonito lets the user explicitly trade reliability against delay by taking into account the charging time distributions.

In terms of practical work, tag-to-tag backscatter communication has mainly focused on physical-layer issues and considers intermittency an orthogonal problem [29, 35, 38, 49]. Instead, the Find neighbor discovery protocol explicitly addresses the intermittency problem and shows that by delaying wake-ups by a random time battery-free nodes can encounter each other faster [19]. We use Find to bootstrap efficient and reliable device-to-device communication with Bonito. Concurrently to our work, a protocol was proposed and implemented that lets devices "die early" when no packet is received to preserve energy and maximize the number of wake-ups [13].

Delay-tolerant networking (DTN). DTN studies networks that are only intermittently connected because of, for example, node failures, mobile users, and power outages [4, 17]. Both DTN and Bonito have the same high-level goal: effective communication in intermittently connected networks. However, DTN and Bonito address orthogonal problems toward the same end goal. While DTN is concerned with forwarding, routing, naming, in-network storage, and optimization of node trajectories to generate encounters in the spatial domain, Bonito aims to generate encounters in the time domain between nodes that are spatially close to each other. Whether concepts from the DTN literature could be applied on top of Bonito is an interesting question for future research.

Energy-aware MAC protocols. Numerous MAC protocols have been proposed for ad-hoc and sensor networks [15]. These protocols turn the radio off most of the time, and power it up only to send or receive a packet. The goal is to achieve a desired network lifetime by maintaining a certain average duty cycle. A fundamental assumption of these protocols is that the radio can be powered up *at any point in time*, which is exploited to reduce idle listening by flexibly scheduling communication among nodes. This is, however, not possible in a battery-free system, where devices are unavailable whenever the capacitor voltage is below a certain threshold, which renders existing energy-aware MAC protocols ineffective.

9 Conclusions

We have presented Bonito, a connection protocol for wireless battery-free devices. By adapting the connection interval to the different and time-varying charging times of intermittently powered nodes, Bonito maintains long-running connections that provide significantly better throughput, latency, and reliability than the state of the art. We have evaluated Bonito by implementing it on a battery-free prototype, conducting testbed experiments with real energy-harvesting traces from diverse scenarios, and demonstrating its utility in an occupancy monitoring case study. With Bonito, we contribute a prime communication primitive, device-to-device unicast, that brings the capabilities of battery-free systems one step closer to those known from today's battery-supported systems.

Availability

The data described in Sec. 2 and a Python implementation of the Bonito protocol from Sec. 3 are available under a permissive MIT license at https://bonito.nes-lab.org/.

Acknowledgments

A Appendix: Gradient Equations

Exponential distribution. The derivative of the log-likelihood function is given by:

$$\mathcal{L}(\lambda) = \log\left(\lambda \cdot \exp(-\lambda x)\right) = \log \lambda - \lambda x_i \qquad (12)$$

$$\nabla \mathcal{L}(\lambda) = \frac{1}{\lambda} - x_i \qquad (13)$$

Calculating the *natural gradient* by defining the step size in terms of the Kullback-Leibler divergence in the distribution space has been shown to speed up convergence in many cases [2]. We obtain the natural gradient by multiplying the regular gradient from (13) with the inverse of the Fisher Information Matrix of the exponential distribution M_{exp}:

$$M_{exp} = \lambda^{-2} \tag{14}$$

$$\frac{\partial \mathcal{L}}{\partial \lambda} = [M_{exp}]^{-1} \cdot \frac{1}{\lambda} - x_i = \lambda - \lambda^2 \cdot x_i \tag{15}$$

Gaussian mixture model. We adopt the gradient equations from [44]: Let $f(x_i, \mu, \sigma^2)$ be the probability density function of the standard normal distribution. The responsibility function $r(x_i, k)$ quantifies the contribution of the k-th component to the model:

$$r(x_i, k) = \frac{\rho_k \cdot f(x_i, \mu_k, \sigma_k^2)}{\sum_l^K \left(\rho_l \cdot f(x_i, \mu_l, \sigma_l^2) \right)} \tag{16}$$

The update equations for the model parameters for the k-th component are then:

$$\frac{\partial \mathcal{L}}{\partial \rho_k} = r(x_i, k) - \rho_k \tag{17}$$

$$\frac{\partial \mathcal{L}}{\partial \mu_k} = \frac{1}{\rho_k} \cdot r(x_i, k) \cdot (x_i - \mu_k) \tag{18}$$

$$\frac{\partial \mathcal{L}}{\partial \sigma_k^2} = \frac{1}{\rho_k} \cdot r(x_i, k) \cdot (x_i - \mu_k)^2 - \sigma_k^2 \tag{19}$$

Normal distribution. We consider the special case of a gaussian mixture model with a single component and also use the equations from [44]:

$$\frac{\partial \mathcal{L}}{\partial \mu} = (x_i - \mu) \tag{20}$$

$$\frac{\partial \mathcal{L}}{\partial \sigma^2} = (x_i - \mu)^2 - \sigma^2 \tag{21}$$

References

[1] Mikhail Afanasov, Naveed Anwar Bhatti, Dennis Campagna, Giacomo Caslini, Fabio Massimo Centonze, Koustabh Dolui, Andrea Maioli, Erica Barone, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, and Luca Mottola. Battery-less zero-maintenance embedded sensing at the mithræum of circus maximus. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems (SenSys)*, 2020.

[2] S. Amari and S.C. Douglas. Why natural gradient? In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 1998.

[3] Abu Bakar and Josiah Hester. Making sense of intermittent energy harvesting. In *Proceedings of the 6th ACM International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems (ENSSys)*, 2018.

[4] Sanjit Biswas and Robert Morris. ExOR: opportunistic multi-hop routing for wireless networks. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM)*, 2005.

[5] Léon Bottou. Stochastic gradient descent tricks. In *Neural Networks: Tricks of the Trade: Second Edition*. Springer Berlin Heidelberg, 2012.

[6] Adriano Branco, Luca Mottola, Muhammad Hamad Alizai, and Junaid Haroon Siddiqui. Intermittent asynchronous peripheral operations. In *Proceedings of the 17th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2019.

[7] Albert Cohen, Xipeng Shen, Josep Torrellas, James Tuck, Yuanyuan Zhou, Sarita Adve, Ismail Akturk, Saurabh Bagchi, Rajeev Balasubramonian, Rajkishore Barik, Micah Beck, Ras Bodik, Ali Butt, Luis Ceze, Haibo Chen, Yiran Chen, Trishul Chilimbi, Mihai Christodorescu, John Criswell, Chen Ding, Yufei Ding, Sandhya Dwarkadas, Erik Elmroth, Phil Gibbons, Xiaochen Guo, Rajesh Gupta, Gernot Heiser, Hank Hoffman, Jian Huang, Hillery Hunter, John Kim, Sam King, James Larus, Chen Liu, Shan Lu, Brandon Lucia, Saeed Maleki, Somnath Mazumdar, Iulian Neamtiu, Keshav Pingali, Paolo Rech, Michael Scott, Yan Solihin, Dawn Song, Jakub Szefer, Dan Tsafrir, Bhuvan Urgaonkar, Marilyn Wolf, Yuan Xie, Jishen Zhao, Lin Zhong, and Yuhao Zhu. Inter-disciplinary research challenges in computer systems for the 2020s. Technical report, 2018.

[8] Alexei Colin, Emily Ruppel, and Brandon Lucia. A reconfigurable energy storage architecture for energy-harvesting devices. In *Proceedings of the 23rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[9] Pablo Corbalán and Gian Pietro Picco. Concurrent ranging in ultra-wideband radios: Experimental evidence, challenges, and opportunities. In *Proceedings of the International Conference on Embedded Wireless Systems and Networks (EWSN)*, 2018.

[10] Stephen Dawson-Haggerty, Andrew Krioukov, Jay Taneja, Sagar Karandikar, Gabe Fierro, Nikita Kitaev, and David Culler. BOSS: Building operating system services. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[11] Jasper de Winkel, Carlo Delle Donne, Kasim Sinan Yildirim, Przemysław Pawełczak, and Josiah Hester. Reliable timekeeping for intermittent computing. In *Proceedings of the 25th ACM International Conference on*

Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2020.

[12] Jasper de Winkel, Vito Kortbeek, Josiah Hester, and Przemysław Pawełczak. Battery-free game boy. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 4(3), 2020.

[13] Vishal Deep, Mathew L. Wymore, Alexis A. Aurandt, Vishak Narayanan, Shen Fu, Henry Duwe, and Daji Qiao. Experimental Study of Lifecycle Management Protocols for Batteryless Intermittent Communication. In *Proceedings of the 18th IEEE International Conference on Mobile Ad Hoc and Smart Systems (MASS)*, 2021.

[14] Farzan Dehbashi, Ali Abedi, Tim Brecht, and Omid Abari. Verification: can wifi backscatter replace RFID? In *Proceedings of the 27th ACM Annual International Conference on Mobile Computing and Networking (MobiCom)*, 2021.

[15] I. Demirkol, C. Ersoy, and F. Alagoz. Mac protocols for wireless sensor networks: a survey. *IEEE Communications Magazine*, 44(4), 2006.

[16] Varick L. Erickson, Miguel Á. Carreira-Perpiñán, and Alberto E. Cerpa. Occupancy Modeling and Prediction for Building Energy Management. *ACM Transactions on Sensor Networks*, 10(3), 2014.

[17] Kevin Fall. A delay-tolerant network architecture for challenged internets. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM)*, 2003.

[18] Kai Geissdoerfer, Mikołaj Chwalisz, and Marco Zimmerling. Shepherd: A portable testbed for the batteryless IoT. In *Proceedings of the 17th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2019.

[19] Kai Geissdoerfer and Marco Zimmerling. Bootstrapping battery-free wireless networks: Efficient neighbor discovery and synchronization in the face of intermittency. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2021.

[20] Graham Gobieski, Brandon Lucia, and Nathan Beckmann. Intelligence beyond the edge: Inference on intermittent embedded systems. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[21] Bernhard Großwindhager, Michael Stocker, Michael Rath, Carlo Alberto Boano, and Kay Römer. Snaploc: An ultra-fast uwb-based indoor localization system for an unlimited number of tags. In *Proceedings of the 18th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, 2019.

[22] Josiah Hester and Jacob Sorber. The Future of Sensing is Batteryless, Intermittent, and Awesome. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems (SenSys)*, 2017.

[23] Sara Khalifa, Mahbub Hassan, Aruna Seneviratne, and Sajal K. Das. Energy-Harvesting Wearables for Activity-Aware Services. *IEEE Internet Computing*, 19(5), 2015.

[24] Meng-Lin Ku, Wei Li, Yan Chen, and K. J. Ray Liu. Advances in energy harvesting communications: Past, present, and future challenges. *IEEE Communications Surveys & Tutorials*, 18(2), 2016.

[25] J.N. Laneman, D.N.C. Tse, and G.W. Wornell. Cooperative diversity in wireless networks: Efficient protocols and outage behavior. *IEEE Transactions on Information Theory*, 50(12), 2004.

[26] Christoph Lenzen, Philipp Sommer, and Roger Wattenhofer. Optimal clock synchronization in networks. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2009.

[27] Tianxing Li and Xia Zhou. Battery-free eye tracker on glasses. In *Proceedings of the 24th ACM Annual International Conference on Mobile Computing and Networking (MobiCom)*, 2018.

[28] Gaosheng Liu and Lin Wang. Self-Sustainable Cyber-Physical Systems with Collaborative Intermittent Computing. In *Proceedings of the 12th ACM International Conference on Future Energy Systems*, 2021.

[29] Vincent Liu, Aaron Parks, Vamsi Talla, Shyamnath Gollakota, David Wetherall, and Joshua R. Smith. Ambient backscatter: Wireless communication out of thin air. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM)*, 2013.

[30] Edward Longman, Oktay Cetinkaya, Mohammed El-Hajjar, and Geoff V. Merrett. Wake-up Radio-enabled Intermittently-powered Devices for Mesh Networking: A Power Analysis. In *Proceedings of the 18th IEEE Annual Consumer Communications Networking Conference (CCNC)*, 2021.

[31] Brandon Lucia, Vignesh Balaj, Alexei Colin, Kiwan Maeng, and Emily Ruppel. Intermittent computing: Challenges and opportunities. In *Proceedings of the 2nd Summit on Advances in Programming Languages (SNAPL)*, 2017.

[32] Brandon Lucia, Brad Denby, Zachary Manchester, Harsh Desai, Emily Ruppel, and Alexei Colin. Computational nanosatellite constellations: Opportunities and challenges. *ACM GetMobile: Mobile Computing and Communications*, 25(1), 2021.

[33] Kiwan Maeng and Brandon Lucia. Adaptive dynamic checkpointing for safe efficient intermittent computing. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[34] Amjad Yousef Majid, Carlo Delle Donne, Kiwan Maeng, Alexei Colin, Kasim Sinan Yildirim, Brandon Lucia, and Przemysław Pawełczak. Dynamic Task-based Intermittent Execution for Energy-harvesting Devices. *ACM Transactions on Sensor Networks*, 16(1), 2020.

[35] Amjad Yousef Majid, Michel Jansen, Guillermo Ortas Delgado, Kasim Sinan Yildirim, and Przemysław Pawełczak. Multi-hop backscatter tag-to-tag networks. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, 2019.

[36] Amjad Yousef Majid, Patrick Schilder, and Koen Langendoen. Continuous sensing on intermittent power. In *Proceedings of the 19th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, 2020.

[37] Giulia Pazzaglia, Marco Mameli, Luca Rossi, Marina Paolanti, Adriano Mancini, Primo Zingaretti, and Emanuele Frontoni. People Counting on Low Cost Embedded Hardware During the SARS-CoV-2 Pandemic. In *Proceedings of Pattern Recognition. ICPR International Workshops and Challenges*, 2021.

[38] Jihoon Ryoo, Yasha Karimi, Akshay Athalye, Milutin Stanaćević, Samir R. Das, and Petar Djurić. Barnet: Towards activity recognition using passive backscattering tag-to-tag network. In *Proceedings of the 16th ACM Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2018.

[39] Muhammad Moid Sandhu, Sara Khalifa, Kai Geissdoerfer, Raja Jurdak, and Marius Portmann. SolAR: Energy positive human activity recognition using solar cells. In *Proceedings of the IEEE International Conference on Pervasive Computing and Communications (PerCom)*, 2021.

[40] Mahadev Satyanarayanan, Wei Gao, and Brandon Lucia. The computing landscape of the 21st century. In *Proceedings of the 20th ACM International Workshop on Mobile Computing Systems and Applications (HotMobile)*, 2019.

[41] Olga Saukh, David Hasenfratz, and Lothar Thiele. Reducing multi-hop calibration errors in large-scale mobile sensor networks. In *Proceedings of the 14th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, 2015.

[42] Lukas Sigrist, Rehan Ahmed, Andres Gomez, and Lothar Thiele. Harvesting-Aware Optimal Communication Scheme for Infrastructure-Less Sensing. *ACM Transactions on Internet of Things*, 1(4), 2020.

[43] Vamsi Talla, Joshua Smith, and Shyamnath Gollakota. Advances and Open Problems in Backscatter Networking. *ACM GetMobile: Mobile Computing and Communications*, 24(4), 2021.

[44] D. M. Titterington. Recursive Parameter Estimation Using Incomplete Data. *Journal of the Royal Statistical Society: Series B (Methodological)*, 46(2), 1984.

[45] Alessandro Torrisi, Kasim Sinan Yildirim, and Davide Brunelli. Enabling Transiently-Powered Communication via Backscattering Energy State Information. In *Applications in Electronics Pervading Industry, Environment and Society*. Springer International Publishing, 2021.

[46] Mathew L. Wymore, Vishal Deep, Vishak Narayanan, Henry Duwe, and Daji Qiao. Lifecycle Management Protocols for Batteryless, Intermittent Sensor Nodes. In *Proceedings of the 39th IEEE International Performance Computing and Communications Conference (IPCCC)*, 2020.

[47] Xun Xian, Xinran Wang, Jie Ding, and Reza Ghanadan. Assisted learning: A framework for multi-organization learning. In *Proceedings of the 34th Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

[48] Kasim Sinan Yildirim and Przemyslaw Pawelczak. On Distributed Sensor Fusion in Batteryless Intermittent Networks. In *Proceedings of the 15th IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS)*, 2019.

[49] Jia Zhao, Wei Gong, and Jiangchuan Liu. X-tandem: Towards multi-hop backscatter communication with commodity wifi. In *Proceedings of the 24th ACM Annual International Conference on Mobile Computing and Networking (MobiCom)*, 2018.

[50] Tongxin Zhu, Jianzhong Li, Hong Gao, and Yingshu Li. Broadcast scheduling in battery-free wireless sensor networks. *ACM Transactions on Sensor Networks*, 15(4), 2019.

Saiyan: Design and Implementation of a Low-power Demodulator for LoRa Backscatter Systems

Xiuzhen Guo
Tsinghua University

Longfei Shangguan
University of Pittsburgh & Microsoft

Yuan He
Tsinghua University

Nan Jing
Yanshan University

Jiacheng Zhang
Tsinghua University

Haotian Jiang
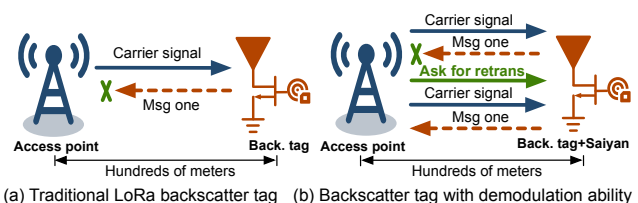*Tsinghua University*

Yunhao Liu
Tsinghua University

(a) Traditional LoRa backscatter tag   (b) Backscatter tag with demodulation ability

Figure 1: Saiyan empowers the LoRa backscatter tag to demodulate feedback signals from a remote access point.

Abstract

The radio range of backscatter systems continues growing as new wireless communication primitives are continuously invented. Nevertheless, both the bit error rate and the packet loss rate of backscatter signals increase rapidly with the radio range, thereby necessitating the cooperation between the access point and the backscatter tags through a feedback loop. Unfortunately, the low-power nature of backscatter tags limits their ability to demodulate feedback signals from a remote access point and scales down to such circumstances.

This paper presents Saiyan, an ultra-low-power demodulator for long-range LoRa backscatter systems. With Saiyan, a backscatter tag can demodulate feedback signals from a remote access point with moderate power consumption and then perform an immediate packet re-transmission in the presence of packet loss. Moreover, Saiyan enables rate adaption and channel hopping – two PHY-layer operations that are important to channel efficiency yet unavailable on long-range backscatter systems. We prototype Saiyan on a two-layer PCB board and evaluate its performance in different environments. Results show that Saiyan achieves 3.5–5× gain on the demodulation range, compared with state-of-the-art systems. Our ASIC simulation shows that the power consumption of Saiyan is around 93.2 μW. Code and hardware schematics can be found at: https://github.com/ZangJac/Saiyan.

1 Introduction

Backscatter radios have emerged as an ultra-low-power and economical alternative to active radios. The ability to communicate over long distances is critical to the practical deployment of backscatter systems, particularly in outdoor scenarios (*e.g.*, smart farm) where backscatter tags need to deliver their data to a remote access point regularly. Conventional RFID technology [12] functions within only a few meters

and is not well suited for outdoor scenarios. To this end, the research community has proposed long-range backscatter approaches [23, 40, 47] that leverage Chirp Spreading Spectrum (CSS) modulation on LoRa [5] to improve the signal resilience to noise, thereby extending the communication range. For instance, LoRa backscatter [47] allows tags to communicate with a source and a receiver separated by 475 m.

However, existing long-range LoRa backscatter systems present a new challenge on packet delivery. The backscatter signals travel twice the link distance and suffer drastic attenuation as the link distance scales. They become very weak after traveling long distances, thereby causing severe bit errors and packet losses. Figure 2 shows the Bit Error Rate (BER) of PLoRa [40] and Aloba [23], two representative long-range LoRa backscatter systems. Evidently, the BER of both systems rises rapidly from less than 1% to over 50% as the tag is moved away from the transmitter (Tx). The receiver is almost unable to demodulate any backscatter signal once the tag is placed 20 m away from the transmitter. Considering that the backscatter tags are unaware of packet loss, each packet must be transmitted blindly for multiple times to lift the packet delivery ratio, which inevitably wastes precious energy and wireless spectrum and cause interference to other radios that work on the same frequency band.

To address these issues, we expect a *downlink* from the access point to the backscatter tag, through which the feedback signals (*e.g.*, asking for a packet re-transmission) can be delivered, thereby forming a *feedback loop*. We envision that such a feedback loop will bring opportunities to bridge

---

Yuan He is the corresponding author.

the gap between long-range backscatter communication and the growing packet loss rate, as reflected on the following aspects:

• *Making on-demand re-transmissions in the presence of packet loss*. The backscatter tag demodulates feedback signals from an access point and makes a re-transmission only if it is asked to do so. This reactive packet re-transmission can mitigate packet loss while improving power and channel efficiency.

• *Scheduling channel hopping to minimize interference*. The unlicensed band where the LoRa resides in is already over crowded. The access point monitors the wireless spectrum and notifies the backscatter tag to switch channels in the presence of in-band interference. As such, the channel utilization and packet delivery ratio can be improved effectively.

• *Adapting data rate to link condition*. The condition of backscatter links varies over time. The access point assesses each backscatter link and keeps the backscatter tag updated through the feedback loop. Each tag then adapts its data rate proactively to utilize the wireless link better.

In addition, such a feedback loop empowers the network administrator to turn on/off sensors on backscatter tags remotely, thereby avoiding labor-intensive and time-consuming physical access to the devices.

To enjoy these benefits, the primary hurdle to overcome is the packet demodulation on LoRa backscatter tags. LoRa is based on frequency modulation. To demodulate a LoRa symbol, the commercial LoRa receiver operates by down-converting the incident LoRa chirp to the baseband, sampling it at twice the chirp bandwidth (BW), and then converting the signal samples from the time domain to the frequency domain using Fast Fourier Transformation (FFT). These operations consume over 40 *mW* power altogether [6]. Considering a miniaturized energy harvester equipped with a palm-sized solar panel, this harvester merely generates 1 *mW* power every 25.4 seconds in a bright day [3]. In other words, to support the standard LoRa demodulation, a backscatter tag needs to wait for 17 minutes until it accumulates enough power. Although the envelope detector has been used for packet demodulation on many backscatter systems [38, 46, 52], it is ill-suited for LoRa demodulation because the envelope of a LoRa signal is a constant.

In this paper, we propose Saiyan, a low-power demodulator for long-range LoRa backscatter systems. Saiyan is based on an observation that *a frequency-modulated chirp signal can be transformed into an amplitude-modulated signal using a differential circuit*. The amplitude of this transformed signal scales with the frequency of the incident chirp signal, thereby allowing us to demodulate a LoRa chirp by tracking the peak amplitude on its transformed counterpart without using power-intensive hardware, such as a down-converter and an ADC. To put this high-level idea into practice, the challenges in design
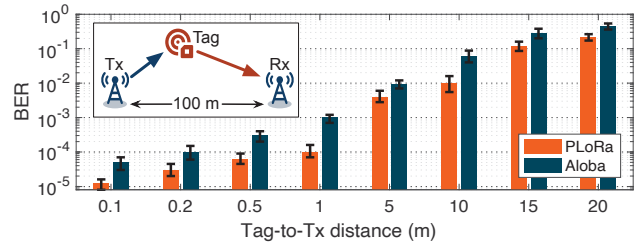

Figure 2: BER of PLoRa [40] and Aloba [23] in different tag-to-transmitter settings. The BER of both systems rises dramatically with the increasing distance between the transmitter and the tag. We re-implement both systems on PCB.

and implementation must be addressed, as summarized below.

Frequency-amplitude transformation. The low-power nature of backscatter tags requires the differential circuit to be extremely low-power. Moreover, to support higher data rate, such a differential circuit should also be hyper-sensitive to the frequency variation of LoRa signals. However, the narrow bandwidth of LoRa signals (*e.g.*, 125/250/500 KHz) renders the conventional detuning circuits, such as RLC resonant circuit, inapplicable. In Saiyan, we instead repurpose the Surface Acoustic Wave (SAW) filter as a signal converter by leveraging its sharp frequency response (§2.1). To minimize the power consumption on demodulation, we further replace the power-consuming ADC with a well-designed double-threshold based comparator (§2.2) coupled by a proactive voltage sampler (§2.3).

Improving the demodulation sensitivity. Although the aforementioned vanilla Saiyan can demodulate LoRa signals with the minimum power consumption, its communication range is limited to 55 m because of the Signal-to-Noise Ratio (SNR) losses in both SAW filter and envelope detector. To extend the communication range, we introduce a low-power cyclic-frequency shifting circuit coupled with an Intermediate Frequency (IF) amplifier to simultaneously remove the noise while magnifying the signal power. This low-power circuit brings 11 dB SNR gain and doubles the demodulation range (§3.1). Furthermore, a low-power correlator is leveraged to extend the demodulation range further to 148 m (§3.2).

Implementation. We implement Saiyan on a 25 *mm*×20 *mm* two-layer Printed Circuit Board (PCB) using analog circuit components and an ultra-low power Apollo2 MCU [13]. The Application Specific Integrated Circuit (ASIC) simulation shows that the power consumption can be reduced to 93.2 *μW*, which is affordable on an energy harvesting tag. The main contributions of this paper are summarized as follows:
• We simplify the standard LoRa demodulation from energy perspective and design the first-of-its-kind low-power LoRa demodulator that can run on an energy harvesting tag.
• We design a set of simple but effective circuits and algorithms, prototyping them on PCB board for system evaluation.

- We demonstrate that Saiyan outperforms the state-of-the-arts on power consumption, communication range, and throughput.

The remainder of this paper is structured as follows. We present the design of vanilla Saiyan in Section 2, followed by super Saiyan in Section 3. Section 4 describes the implementation details. The experiment (§5) follows. We review related works in Section 6 and conclude in Section 7.

2 Vanilla Saiyan

A LoRa symbol is represented by a chirp whose frequency grows linearly over time, as formulated below.

$$s(t) = A sin(2\pi f(t)t) \tag{1}$$

where A is the signal amplitude; $f(t) = F_0 + kt$ describes how the frequency of this chirp signal varies over time; F_0 is the initial frequency offset; k is the frequency changing rate. The frequency of a LoRa chirp wraps to 0 right after peaking BW—the bandwidth of LoRa. Different LoRa chirps peak the frequency BW at different time due to the difference in their initial frequency offset, as shown in Figure 3(a). Applying a differential to a LoRa chirp, we have:

$$s'(t) = \frac{ds(t)}{dt} = A cos(2\pi f(t)t)[2\pi \frac{df(t)}{dt} t + 2\pi f(t)] \tag{2}$$
$$= \underbrace{2\pi A(F_0 + 2kt)}_{Amplitude} cos(2\pi f(t)t)$$

The above equation indicates that the amplitude of the transformed signal $s'(t)$ is proportional to the frequency of the input LoRa chirp $s(t)$, as shown in Figure 3(b). This frequency-amplitude correlation allows us to demodulate the frequency-modulated (FM) chirp signal by tracking the peak amplitude of its transformed amplitude-modulated (AM) counterpart.

2.1 Frequency-amplitude Transformation

To realize the differential operation [10], an intuitive solution is using RLC resonant circuit. However, the narrow bandwidth of LoRa (*e.g.*, 125/250/500 KHz) renders this idea infeasible (see Appendix A.1 for details). In Saiyan we instead exploit the sharp frequency response of the Surface Acoustic Wave (SAW) filter to transform LoRa chirps into amplitude-modulated signals.

SAW filter primer. A SAW filter consists of two interdigital transducers (shown in Figure 4). The input interdigital transducer transforms electrical signals into acoustic waves; the output interdigital transducer then transforms acoustic waves back into electrical signals. This two-stage signal transformation introduces 6 dB insertion loss to the incident signal [4].

Re-purposing SAW filter as a signal converter. Our design is based on the observation that the frequency response of a
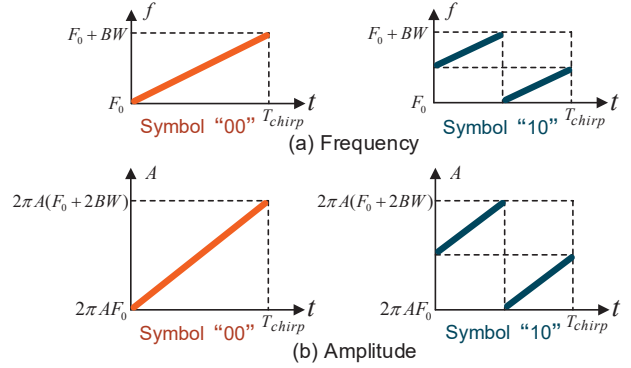

Figure 3: LoRa symbols before and after frequency-amplitude transformation. (a) Different LoRa symbols in the frequency domain. (b) The amplitude of LoRa symbols after frequency-amplitude transformation.
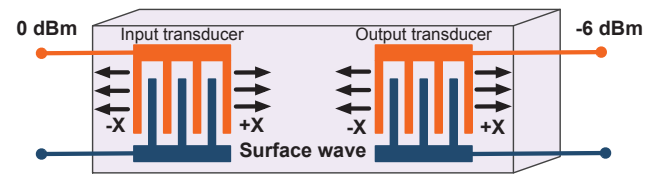

Figure 4: Diagram of the SAW filter. The SAW filter converts electrical signal into acoustic signal and then back through two inter-digital transducers.

SAW filter grows monotonically within a certain frequency band (termed as *critical band*). After passing through the critical band, the chirp signal will be transformed into an AM signal whose amplitude scales with the frequency of this input FM chirp. This allows us to demodulate LoRa chirp by simply tracking the peak amplitude of the AM signal. On the other hand, since SAW filter by its own design is battery-free, such frequency-amplitude transformation doesn't incur extra power consumption to backscatter tags.

In Saiyan, we take into account the working frequency and bandwidth of LoRa signals and select a general-purpose Qualcomm B3790 [1] SAW filter as the signal converter. Figure 5 shows its frequency response. The signal amplitude grows by 25 dB as the frequency of the incident signal scales from 433.5 MHz to 434 MHz (500 KHz bandwidth). To validate this 25 dB amplitude gap is strong enough for differentiating LoRa chirps, we feed four different chirp symbols into this SAW filter and plot the output in Figure 6. Evidently, these symbols peak their amplitude at clearly different time points, confirming the effectiveness of the SAW filter.

2.2 Demodulation

The transformed symbols are down-converted to the baseband through an envelope detector. Before demodulation, the
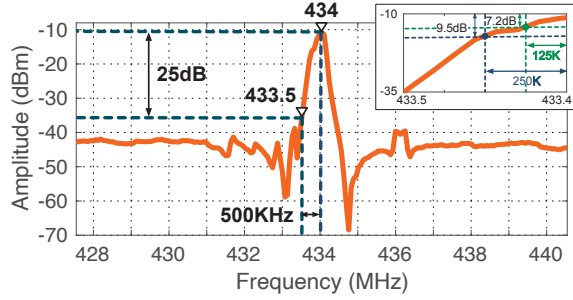
Figure 5: The amplitude-frequency response of the SAW filter adopted by Saiyan. The central frequency is 434 MHz. The measured insertion loss of this SAW filter is 10 dB. We observe 25 dB, 9.5 dB, and 7.2 dB amplitude variation as the frequency of an incident signal grows from 433.5 MHz, 433.75 MHz, and 433.875 MHz to 434 MHz, respectively.

standard LoRa receiver first digitizes these baseband signals using an ADC, which is power intensive. To save power, an intuitive solution is to replace the ADC with a low-power voltage comparator [37, 46]. The threshold of this comparator is set to a value slightly lower than the peak amplitude of the basband signal. This allows us to locate the peak amplitude by checking the comparator's output. Unfortunately, due to the in-band interference and hardware noise, the transformed AM signal may experience multiple amplitude peaks and valleys that may confuse the comparator.

Double-threshold based comparator. In Saiyan we instead adopt a double-threshold based comparator to stabilize the output binary sequence. Let U_H and U_L denote the high-voltage and low-voltage threshold defined in this comparator. When the amplitude of an input signal is sufficiently higher than U_H, the comparator outputs a high voltage. When the amplitude of this signal is equivalent to U_H or above, no chattering occurs since the output will not respond unless the input falls below U_L. Following this idea, the output voltage B_i can be formulated as follows:

$$B_i = \begin{cases} low, & if \quad A_i < U_H \quad \& \quad B_{i-1} = low \\ high, & if \quad A_i \geq U_H \quad \& \quad B_{i-1} = low \\ low, & if \quad A_i < U_L \quad \& \quad B_{i-1} = high \\ high, & if \quad A_i \geq U_L \quad \& \quad B_{i-1} = high \end{cases} \quad (3)$$

where A_i represents the amplitude of the ith signal sample. This double-threshold comparator takes into account the amplitude samples both in the past and present. It nulls out the chattering caused by the amplitude oscillation. The threshold setup is detailed in system implementation (§4).

To show the effectiveness of this double-threshold based comparator, we apply it to a LoRa chirp and plot the output in Figure 7. For comparison, we also plot the output of two single-threshold based comparators (U_H alone and U_L alone, respectively). We can see that using a high threshold U_H

Table 1: The required sampling rate (KHz) in theory/practice to achieve 99.9% decoding accuracy.

	SF=7	SF=8	SF=9	SF=10	SF=11	SF=12
K=1	15.6/20	7.8/12	3.9/5.5	1.95/2.6	0.98/1.2	0.49/0.6
K=2	31.2/40	15.6/20	7.8/12	3.9/5.5	1.95/2.6	0.98/1.2
K=3	62.5/85	31.2/40	15.6/20	7.8/12	3.9/5.5	1.95/2.6
K=4	125/180	62.5/85	31.2/40	15.6/20	7.8/12	3.9/5.5
K=5	250/400	125/180	62.5/85	31.2/40	15.6/20	7.8/12

alone fails to detect the amplitude peak due to the valleys emerging on signal amplitude (*i.e.*, $t \in [t_E, t_F]$ in Figure 7(b)). Using a low threshold U_L alone causes false positives due to the misleading peak emerging on the signal amplitude ($t \in [t_A, t_B]$ in Figure 7(d)). In contrast, the double-threshold based comparator produces a series of stable binary voltages that can guide us to locate the peak amplitude at the correct position (*i.e.*, at the tail of the high voltage samples t_F shown in Figure 7(e)).

2.3 Low-power Voltage Sampler

The comparator quantizes chirp samples into binary voltages which are stored in a counter of micro-controller (MCU). The sampling rate tradeoffs the power consumption and the demodulation performance and thus cannot be set arbitrarily. A higher sampling rate supports a higher link throughput. It however consumes more power. Suppose a LoRa chirp encodes K bits data. The data rate equals $K \cdot BW / 2^{SF}$, where BW and SF respectively represent bandwidth and spreading factor. According to the Nyquist sampling theorem, the sampling rate should be not lower than $2 \cdot BW / 2^{SF-K}$.

However, in reality, using the theoretical minimum sampling rate exacerbates bit errors. We conduct a benchmark experiment to measure the practical sampling rate required to achieve 99.9% decoding accuracy. Table 1 lists the results with different settings of spreading factor and coding rate. We find that the required sampling rate in practice is slightly higher than the theoretical minimum sampling rate. Suggested by this result, we conservatively set the sampling rate to $3.2 \cdot BW / 2^{SF-K}$, which guarantees the demodulation performance.

Decoding. After quantization, the low-power MCU decodes each LoRa chirp by localizing the bit '1' within each LoRa symbol, as shown in Figure 8. The LoRa preamble contains ten identical up-chirps. Upon detecting the LoRa preamble, Saiyan waits for 2.25 symbol times (sync. symbols) and operates demodulation on the payload hereafter.

Remarks. The vanilla Saiyan demodulates LoRa signals with the minimum power consumption. However, its demodulation sensitivity is limited due to the signal attenuation in the SAW
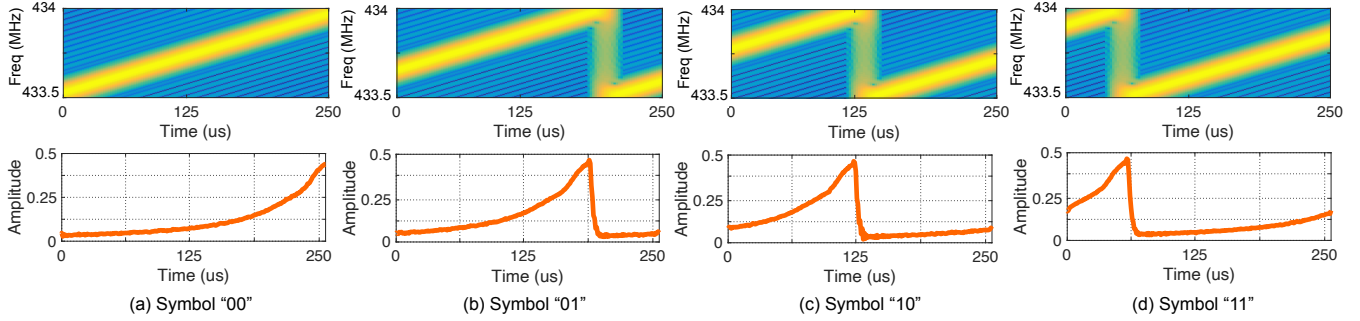
Figure 6: The input (top) and output (bottom) signals of the SAW filter. The amplitude of the output signal scales with the frequency of the input signal. They reach the maximal value simultaneously.
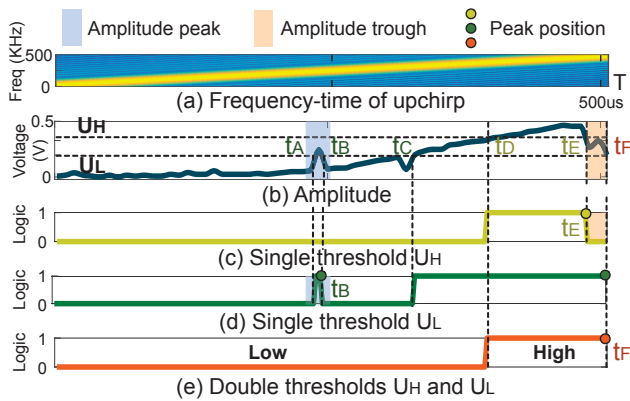

Figure 7: Comparing the output of different voltage comparators. (a): the incident LoRa chirp. (b): the output of an envelope detector. (c)-(d): the output of the single-threshold based comparator that uses U_H or U_L as the cut-off amplitude. (e) the output of the double-threshold based comparator that uses U_H and U_L simultaneously as the cut-off amplitudes.

filter and the noise added by the envelope detector. Next, we introduce super Saiyan to improve the sensitivity.

3 **Super** Saiyan

Super Saiyan takes the following actions to consistently improve the demodulation sensitivity: *i*) improving the SNR of baseband chirp signals with a cyclic-frequency shifting circuit, and *ii*) improving the sensitivity of demodulator with correlation.

3.1 Cyclic-frequency Shifting

Understanding the principle of envelope detector. The envelope detector has been widely adopted by low-power RF devices to down-convert the incident signal. However, due to the inherent non-linearity caused by the squaring operation of
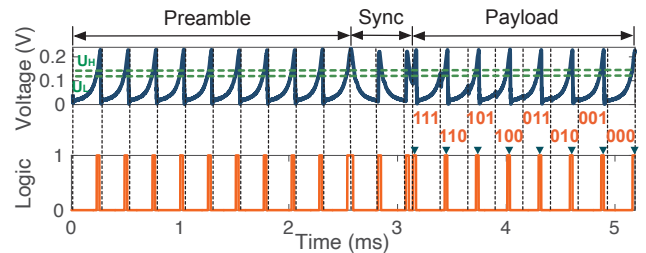

Figure 8: The decoding process of a LoRa packet

CMOS devices [27], both the targeted signal (*i.e.*, feedback signals from the LoRa access point) and the RF noises will be down-converted to the baseband. Consequently, the targeted signal becomes even weaker after down-conversion. We explicate this phenomenon using the following example. Let S_{in} be the incident signal: $S_{in} = S_t + S_n$, where S_t and S_n denote the targeted signal and RF noises, respectively. The output signal S_{out} of this envelope detector can be represented by:

$$S_{out} = kS_{in}^2 = k(S_t + S_n)^2 \qquad (4)$$
$$= kS_t^2 + 2kS_t \cdot S_n + kS_n^2$$

where k represents the attenuation factor. The first term S_t^2 on the right side of this equation manifests that the targeted signal S_t is shifted to the baseband through self-mixing. The second and the third terms both indicate the RF noises are shifted to the baseband after mixed with the targeted signal and the noises themselves, respectively, causing strong interference on the baseband.

Cyclic-frequency shifting. In Saiyan we design a low-power circuit to mitigate the SNR loss brought by the envelope detector. The circuit is realized by two RF mixers and two clock signals. Its operation is detailed as follows.

• **Step 1**. The micro-controller first generates a clock signal $CLK_{in}(\Delta f)$ and mixes it with the incident signal $S(F)$, resulting in two sideband signals $S(F - \Delta f)$ and $S(F + \Delta f)$, as shown in Figure 9(a)-(b). The sideband signals and the
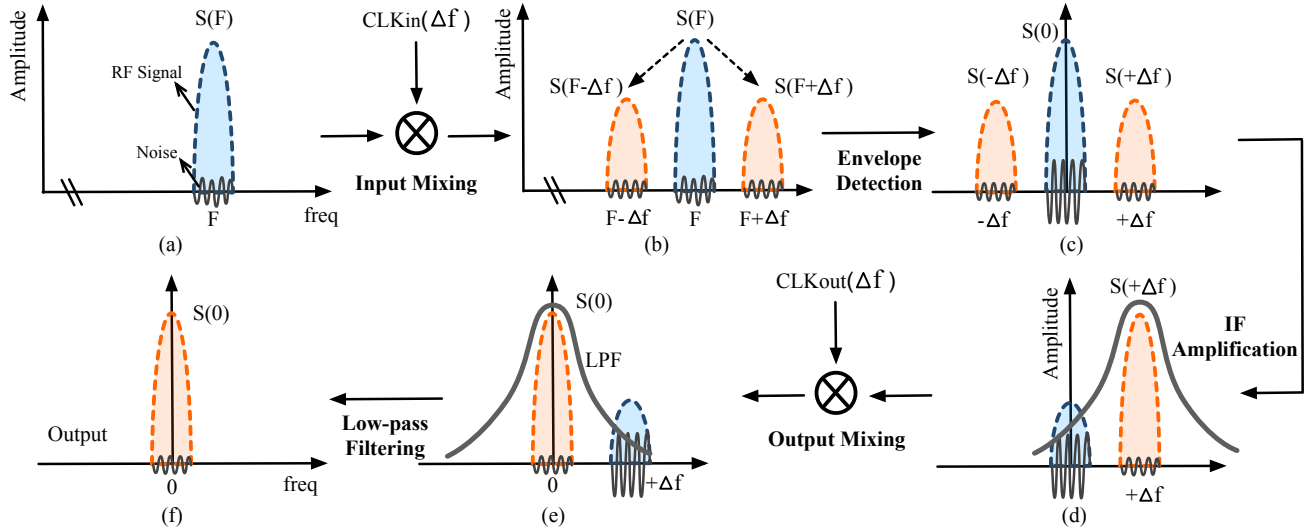
Figure 9: The illustration of the cyclic-frequency shifting. (a) The input signal $S(F)$. (b) $S(F)$ is first mixed with the clock signal, resulting in two sideband signals $S(F-\Delta f)$ and $S(F+\Delta f)$. (c) The envelope detector extracts the envelope of those three signals and down-converts them to the the baseband. (d) The IF amplifier boosts the power of $S(\Delta f)$ and attenuates the power at other frequency bands. (e) The desired signal $S(\Delta f)$ with significantly lower noises is shifted back to the baseband. (f) The output signal $S(0)$.

incident signal are then down-converted to the intermediate frequency (IF) band (denoted by $S(-\Delta f)$ and $S(\Delta f)$) and the baseband (denoted by $S(0)$) respectively with an envelope detector (Figure 9(c)).

• **Step 2**. Since RF noises are not down-converted to the IF band by the envelope detector, we amplify the unpolluted IF signal $S(\Delta f)$ using a low-power IF amplifier. The frequency selectivity of this IF amplifier filters out signals at other frequencies (*e.g.*, $S(0)$), as shown in Figure 9(d).

• **Step 3**. The power-amplified IF signal $S(\Delta f)$, mixed with another clock signal $CLK_{out}(\Delta f)$, is shifted back to the baseband, as shown in Figure 9(e). At the same time, the noisy baseband signal $S(0)$ will be shifted to the IF band and then filtered by a low-pass filter (Figure 9(f)).

In a nutshell, this circuit first moves the targeted signal to an intermittent frequency band (step 1) to avoid the RF noise contamination introduced by the envelope detector. This also leaves us an opportunity to remedy the SNR loss in down-conversion (step 2). Finally, the targeted signal is moved back to the baseband for demodulation. At the same time the DC offset, flicker and other noises are moved to the IF band and removed by a low-pass filter (step 3).

Figure 10 shows the spectrums before and after feeding the chirp signal into the cyclic frequency shifting circuit. Evidently, both the inband and out-of-band RF noises have been cleaned by the circuit, ensuring the decodability of chirp signals. Our quantitative measurement shows that the cyclic-frequency shifting circuit brings in 11 dB SNR gain.

Clock signal generation. The above circuit design relies on two clock signals $CLK_{in}(\Delta f)$ and $CLK_{out}(\Delta f)$. To save power,
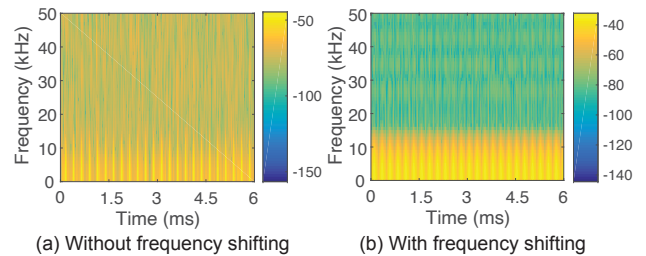

Figure 10: The spectrum of an incident LoRa signal when being down-converted into the baseband with an envelope detector. (a) Without cyclic-frequency shifting. (b) With cyclic-frequency shifting. The LoRa signal contains 24 LoRa chirps (*BW*=500KHz, *SF*=8).

we program the MCU to generate $CLK_{in}(\Delta f)$ signal and then leverage a delay line to copy $CLK_{in}(\Delta f)$ as $CLK_{out}(\Delta f)$:

$$CLK_{out}(\Delta f) = CLK_{in}(\Delta f + \Delta \phi) \tag{5}$$

where $\Delta \phi$ is the phase shift caused by the delay line. We tune the length of this delay line to ensure $cos(\Delta \phi) \approx 1$ so that $CLK_{out}(\Delta f)$ equals $CLK_{in}(\Delta f)$.

Circuit integration. We integrate this cyclic-frequency shifting circuit into the envelope detector. Figure 11 shows the schematic of this design. It consists of an input mixer, an output mixer, an envelope detector, an IF amplifier, a low-pass filter (LPF), an oscillator, and a transmission line. Specifically, The base clock signal is provided by a micro-power precision oscillator LTC6907 [11]. A low-power transistor 2N222 [8] is adopted as the IF amplifier.
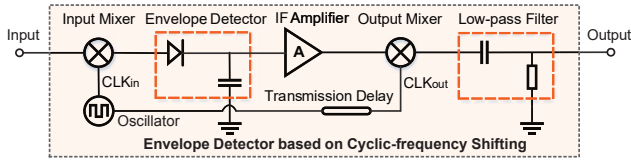
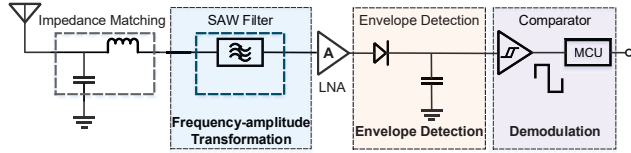Figure 11: The schematic of cyclic-frequency shifting.

Figure 12: The high-level circuit schematic of Saiyan.

3.2 Correlation

While the above cyclic-frequency shifting circuit successfully improves the SNR of the incident signal, the demodulation accuracy still suffers degradation when the incident signal is too weak, *e.g.*, close to the noise floor. We thus employ correlation — a mainstream approach that has been largely adopted for packet detection to further improve the demodulation sensitivity. It operates by correlating signals samples with a local chirp template. An energy peak shows up as long as the incident signal matches the template. The receiver then tracks the energy peak and demodulates the incident signal.

4 Implementation

We describe the system implementation in this section.

4.1 Backscatter Tag

We implement Saiyan on a 25 *mm*×20 *mm* two-layer PCB using commercial off-the-shelf analog components and an ultra-low power Apollo2 (10 *μ*A/MHz) [13] MCU. We determine its size through a mixed analytical and experimental approach, striking a balance between the form factor and circuit interference. Figure 13 shows the hardware prototype. Saiyan functions with an omni-directional antenna [2] with 3 dBi gain.

Architecture and workflow. Figure 12 shows the architecture of Saiyan. The incident signal passes through a passive SAW chip B39431B3790Z810 [1] and is transformed into an amplitude-modulated signal. We place a common-gate low-noise amplifier (CGLNA) [17] between the SAW filter and the customized envelope detector to amplify the transformed signal. The amplified signal is then down-converted to the baseband through the envelope detector. Finally, a low-power voltage comparator NCS2202 [9] is leveraged to quantize the output signal from the envelope detector.
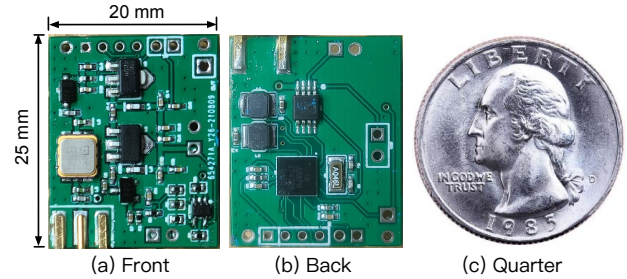

Figure 13: The hardware prototype of Saiyan. The quarter next to Saiyan demonstrates the form factor.

Plug-and-play. As an ultra-low-power peripheral, Saiyan can be integrated into the existing long-range LoRa backscatter systems [23, 40] with ignorable engineering efforts. Taking PLoRa [40] as an example, we replace its packet detection module with Saiyan and retained all the remaining functional units the same. This simple replacement allows PLoRa tag to demodulate the feedback signals while retaining the modulation capability at the same time. On the software side, we replicate the sampling rate control logic to facilitate the demodulation.

Power management. The energy harvester on Saiyan comprises of a palm-sized photovoltaic panel and a high-efficiency step-up DC/DC converter LTC3105 [3]. It generates 1 *mW* power every 25.4 seconds in a bright day. The power management module provides a constant 3.3V output voltage to the MCU. The power consumption of this power management module in working mode is approximately 24 *μW*

Determining the voltage thresholds U_H and U_L. Ideally, U_H should be slightly lower than the peak amplitude of the input signal A_{max}. Let G be the gap between A_{max} and the voltage threshold U_H. We have: $G = 20lg(A_{max}/U_H)$. Thus, U_H can be estimated on the basis of the following equation: $U_H = A_{max}/10^{\frac{G}{20}}$. The threshold voltage U_L is set to $U_H - U_F$, where U_F represents the amplitude of the envelope detector's output. The thresholds U_H and U_L are tuned by two adjustable on-board resistors. In practice, considering that A_{max} and U_F both vary with the link distance, we measure these two values offline under different link distance settings and store a mapping table on each tag to facilitate the configuration of U_H and U_L. To alleviate this manual configuration overhead, one could leverage an Automatic Gain Control (AGC) [42, 43] to adapt the power gain automatically. We leave it for future work.

4.2 LoRa Transmitter and Receiver

LoRa transmitter. We use two types of LoRa transmitters in the evaluation: *i*) a LoRa transmitter implemented on a software-defined radio platform USRP N210, and *ii*) a com-

Figure 14: Outdoor experiment field.

mercial off-the-shelf LoRa node equipped with a Semtech SX1276RF1JAS [7] chip. Both platforms use a single omnidirectional antenna with 3 dBi gain. The transmission power is set to 20 dBm.

LoRa receiver. The LoRa receiver is implemented on a software-defined radio platform USRP N210. We set the sampling rate to 10 MHz, thereby allowing the receiver to monitor six LoRa channels simultaneously.

4.3 ASIC Simulation

We simulate the Application Specific Integrated Circuit (ASIC) of Saiyan based on the TSMC 65-nm CMOS process. The active area of on-chip Integrated Circuits (IC) is $0.217 \ mm^2$. The ASIC simulation shows that the power consumption of Saiyan is 93.2 μW. Specifically, the power consumption of LNA, oscillator, and digital circuit is 68.4 μW and 22.8 μW, and 2 μW, respectively. Once Saiyan demodulated the feedback signals, the MCU starts preparing data for packet re-transmissions, which consumes extremely low power (*i.e.*, the power consumption of the ultra-low power Apollo2 [13] in Saiyan is merely 19.6 μW).

4.4 MAC-layer for Multi-tag Coexistence

We briefly discuss MAC-layer in this section. The downlink packets can be divided into three groups: unicast packet, multicast packet, and broadcast packet. In unicast, all backscatter tags within the radio range will receive and demodulate this unicast packet from the access point. However, only the targeted tag will response (*e.g.*, re-transmit the lost packet). Hence, no collision occurs. However, in multicast and broadcast, collision happens as long as more than one backscatter tag replies at the same time. For instance, the access point sends a downlink packet (*e.g.*, turn off the humidity sensor), while multiple tags acknowledge the reception of this downlink packet simultaneously. In this case, the access point can leverage slotted ALOHA [22] protocol to coordinate tags and minimize collisions. We take Figure 15 as an example to illustrate the MAC-layer operation. Suppose three tags are sending an acknowledgement to the access point to confirm the reception of a downlink packet. Each tag will randomly select a time slot and store it in its local counter. Upon the
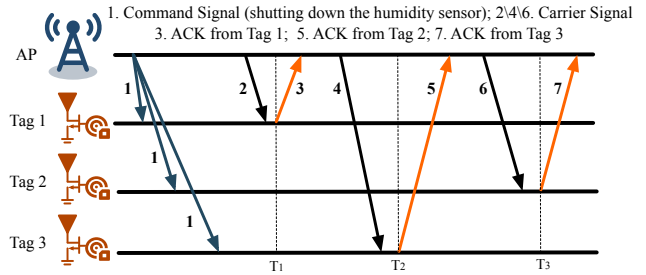

Figure 15: The illustration of MAC-layer operations in Saiyan. Each tag randomly selects a slot to transmit. The access point (AP) signals the beginning of each slot with a carrier signal.

detection of a carrier signal from the access point, each tag decreases the slot number by one and transmits as soon as the slot number goes zero. The randomness in slot selection minimizes the interference among tags.

5 Evaluation

In this section, we present the evaluation results of field studies (§5.1) and micro-benchmarks (§5.2). Two case studies follow (§5.3). Unless otherwise posted, the transmitter and the receiver are collocated throughout the experiment.

Setups. The LoRa transmitter works on the 433.5 MHz frequency band. The spreading factor and the bandwidth are set to 7 and 500 KHz, respectively. The payload of each LoRa packet contains 32 chirp symbols. In each experiment, we let the transmitter transmit 1,000 LoRa packets and then repeat the experiment for 100 times to ensure the statistical validity. We adopt *BER*, *throughput*, and *demodulation range* as the key metrics to assess Saiyan's performance.

- **BER** refers to the ratio of error bits to the total number of bits received by Saiyan.

- **Throughput** measures the amount of received data correctly decoded by Saiyan within one second.

- **Demodulation range** refers to the maximum distance between the tag and the LoRa transmitter when the BER is maintained below 1‰.
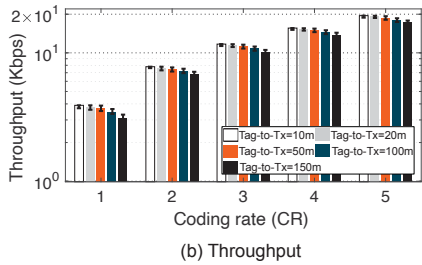
5.1 Field Studies

We conduct field studies both indoors and outdoors to assess the impact of coding rate (CR), spreading factor (SF), and bandwidth (BW) on BER, demodulation range, and throughput, which are three key evaluation metrics.
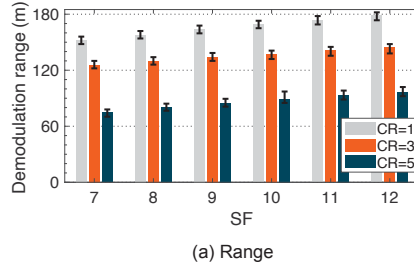
5.1.1 Outdoor experiments

Impact of coding rate. We place a Saiyan tag 10 m, 20 m, 50 m, 100 m, and 150 m away from a LoRa transmitter. Under
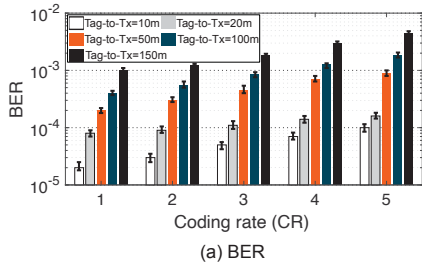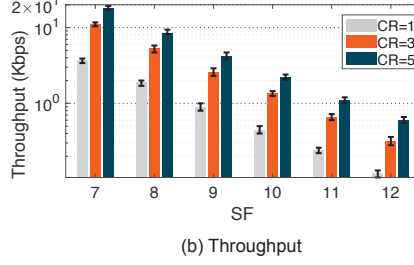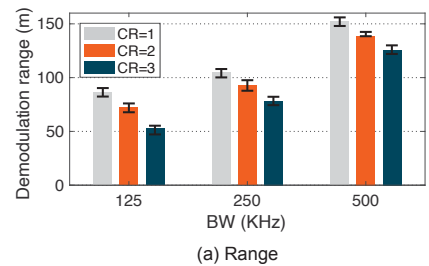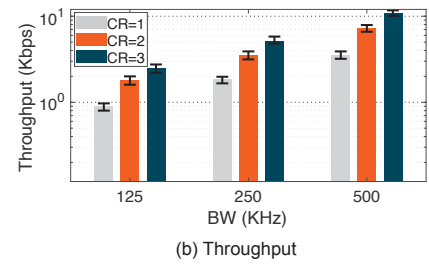
Figure 16: BER and throughput in different coding rate settings.

Figure 17: Demodulation range and throughput in different SF settings.

Figure 18: Demodulation range and throughput in different BW settings.

each distance setting, we vary the coding rate of LoRa signals and measure BER and throughput. We have three observations based on the results shown in Figure 16.

First, the BER grows with the coding rate. As shown in Figure 16(a), the BER under the highest coding rate setting (*i.e.*, 5) is 2.4–5.2× higher than the BER under the lowest coding rate setting (*i.e.*, 1) across all different Tx-to-tag distances. For instance, when the Tx-to-tag distance is 100 m, Saiyan achieves a BER of 1.85‰ under the highest coding rate setting. The BER then drops to 0.4‰ under the same Tx-to-tag distance setting when we change the coding rate to 1. This is expected since the Saiyan tag has to differentiate more types of LoRa chirps under the high coding rate setting.

Second, the throughput grows linearly with the coding rate (Figure 16(b)). For example, when the Tx-to-tag distance is 100 m, the achievable throughput at CR=5 (18.12 Kbps) is around 5.1× higher than the throughput at a coding rate of 1 (3.57 Kbps).

Third, both the BER and the throughput get exacerbated with the growing Tx-to-tag distance. For instance, when CR=5, the BER grows dramatically from 0.1‰ to 4.4‰ as the Tx-to-tag distance grows from 10 m to 150 m. The throughput, on the other hand, declines from 19.6 Kbps to 17.2 Kbps. This is expected since Saiyan relies on the signal power to demodulate the incident LoRa signal.

Impact of spreading factor. Next, we vary the spreading factor from 7 to 12 and assess Saiyan's demodulation range and throughput under each setting. The results are shown in Figure 17. We observe that the demodulation range grows with the increasing spreading factor. The throughput, on the contrary, declines with the increasing spreading factor. For instance, the demodulation range under the highest spreading factor setting (*i.e.*,SF=12) is 1.1–1.3× longer than the

demodulation range under the lowest spreading factor setting (*i.e.*, SF=7) across three different coding rate settings. The throughput drops by 30.3–35.1× as we decrease the SF from 12 to 7. This is expected since a higher spreading factor enhances the anti-noise capability of LoRa signals; thus the demodulation range grows. On the other hand, the symbol time grows with the increasing spreading factor, resulting in a lower throughput.

Impact of bandwidth. We set the spreading factor to 7 and assess the impact of LoRa bandwidth on the demodulation range and throughput. The results are shown in Figure 18. We observe that the demodulation range and the throughput both grow with the LoRa bandwidth. Specifically, given the coding rate of 2, the demodulation range grows from 72.2 m to 138.6 m as we increase the bandwidth from 125 KHz to 500 KHz. On the other hand, since the LoRa symbol time is inversely proportional to the bandwidth, we observe the throughput drops around 4× from 7.2 Kbps to 1.8 Kbps as we decrease the bandwidth from 500 KHz to 125 KHz.

5.1.2 Indoor experiments

We repeat the above experiments in an indoor environment where the LoRa signals have to penetrate one or multiple concrete walls to arrive at the backscatter tag.

Penetrating one concrete wall. Similar to the trend shown in the outdoor scenario, the throughput measured in the indoor scenario also grows with the increase of the coding rate (Figure 19). For example, the throughput grows from 3.7 Kbps to 18.7 Kbps when the coding rate varies from 1 to 5. The demodulation range, on the other hand, declines from 48.8 m to 26.2 m as we increase the coding rate from 1 to 5.

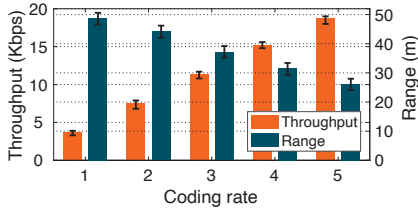**Penetrating two concrete walls**. The LoRa signal experi-

Figure 19: Throughput and downlink range in the presence of one concrete wall.
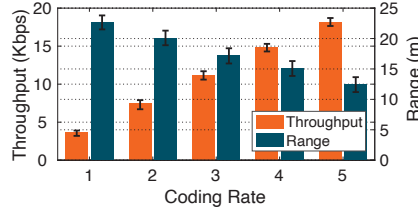

Figure 20: Throughput and downlink range in the presence of two concrete walls.
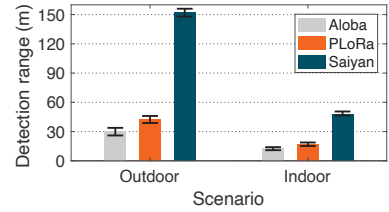

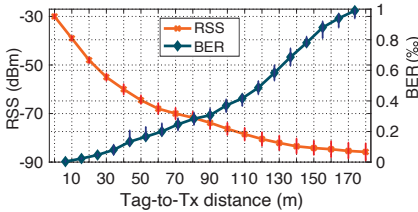Figure 21: Comparison of Saiyan, Aloba, and PLoRa on the detection range.
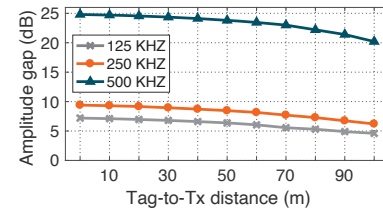

Figure 22: RSS and BER over distance.

Figure 23: The amplitude gap of the output signal after SAW filter
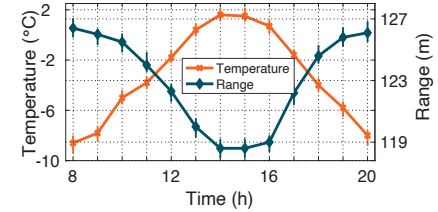

Figure 24: Demodulation range under different temperatures

ences stronger attenuation when penetrating two concrete walls. Accordingly, we observe the demodulation range and the throughput decline by 2.21-2.09× and 1.01-1.05× compared to those under the single concrete wall settings (Figure 20).

5.1.3 Comparison with state-of-the-art systems

We further compare Saiyan with two state-of-the-art systems, namely, Aloba [23] and PLoRa [40] in both outdoor and indoor environments. PLoRa operates cross-correlation to detect a LoRa packet. Aloba feeds the incident signal into a moving average filter and then leverages the unique RSSI pattern of the LoRa preamble to detect a LoRa packet. They both cannot demodulate the payload. Therefore, we compare them with Saiyan in terms of the packet detection range.

Figure 21 shows the experiment result. In the outdoor line-of-sight settings, Saiyan achieves a packet detection range of 148.6 m, outperforming ALoBa (30.6m) and PLoRa (42.4m) by 4.52× and 3.26×, respectively. In an indoor none-line-of-sight environment, although the packet detection range of Saiyan declines to 44.2 m, it still outperforms Aloba (12.4 m) and PLoRa (16.8 m) by 3.56× and 2.63×, respectively.

5.2 Micro-benchmarks

To better understand the performance of each design component in Saiyan, we run micro-benchmarks to assess the receiver sensitivity, the SAW filter, as well as the power consumption and the system cost.

5.2.1 Receiver sensitivity

We define the receiver sensitivity as the minimum Received Signal Strength (RSS) of an incident signal that can be detected by Saiyan. To assess the receiver sensitivity, we measure the BER and the Received Signal Strength (RSS) under different Tx-to-tag distance settings. As expected, the BER grows gradually with the increase of the Tx-to-tag distance, as shown in Figure 22. Nevertheless, Saiyan can still detect the incident signal when the tag is 180 m away from the transmitter. As we increase the tag-to-Tx further, the signal strength is too weak to be detected by Saiyan. The above experiment demonstrates an -85.8 dBm receiver sensitivity, outperforming the conventional envelope detector by 30 dBm [27].

5.2.2 Performance of the SAW filter

Frequency-amplitude response. Saiyan relies on the frequency-amplitude response of the SAW filter to demodulate LoRa signals. A sharp frequency-amplitude response (*e.g.*, a small frequency variation leads to a large amplitude gap) is desirable as it allows the Saiyan tag to detect the minute frequency variation on the incident signal.

We feed LoRa signals with different bandwidth into the SAW filter and measure the amplitude variation of the output signal. The results are shown in Figure 23. As expected, the amplitude variation of the output signal (*a.k.a.*, amplitude gap) tends to be less significant with the decreasing chirp bandwidth. For instance, when the Tx-to-tag distance is 10 m, the amplitude gap drops from 24.7 dBm to 9.3 dBm, and further to 7.1 dBm as we decrease the chirp bandwidth from 500 KHz to 250 KHz, and further to 125 KHz, respectively. A similar trend shows up as we increase the Tx-to-tag distance.
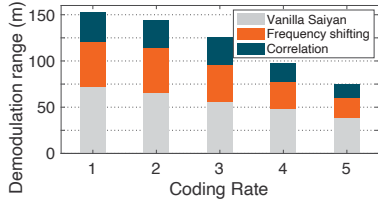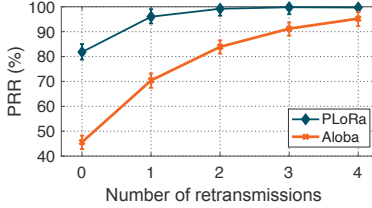
Figure 25: Ablation study of Saiyan.
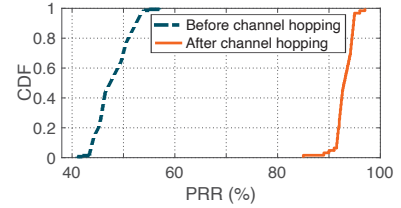

Figure 26: PRR in different settings.

Figure 27: PRR lifts with Saiyan.

Table 2: Energy consumption (under 1% duty cycling) and cost of each component in Saiyan tag.

Component	SAW Filter	LNA	OSC Clock	Envelope Detector	Comparator	MCU	Total
Energy (μW)	0	248.5	86.8	0	14.45	19.6	369.4
Cost ($)	3.87	4.15	1.25	1.20	1.26	15.43	27.2

For instance, when the signal bandwidth is 500 KHz, the amplitude gap of the output signal drops from 24.7 dBm to 20.2 dBm as the Tx-to-tag distance increases from 10 m to 100 m.

The impact of temperature. The frequency selectivity of the SAW filter is affected by the ambient temperature [36]. We thus run an experiment to assess the impact of temperature on the demodulation range. The experiment is conducted outdoors on a sunny day from 8 a.m. to 8 p.m.. Figure 24 shows the result. We observe that the demodulation range in general is insensitive to the temperature. For instance, when the temperature rises from the lowest -8.6 °C at 8 a.m. to the highest 1.6 °C at 2 p.m., the demodulation range merely drops from 126.4 m to 118.6 m.

5.2.3 Ablation study

We conduct an ablation study to assess the effectiveness of each design component of Saiyan. In this experiment, we set the spreading factor and the bandwidth to 7 and 500 KHz respectively and measure the maximum demodulation range under different coding rate settings. The results are shown in Figure 25. We find that the vanilla Saiyan achieves a relatively short demodulation range (38.4 m—72.6 m) across five different coding rate settings. The demodulation range then grows by 1.56×–1.73× with the help of the cyclic frequency shifting module. The cross-correlation further improves the demodulation range by 1.94×–2.25×.

5.2.4 Power consumption & system cost

Table 2 summarizes the power consumption (under 1% duty cycling as in LoRa [22]) and cost of each component in Saiyan. Among these hardware components, the most power-hungry parts are LNA and oscillator (OSC) clock, which account for 67.3% and 23.5% of the total power consumption, respectively. As we demonstrate in §4.3, the power consumption can be effectively reduced by 74.8% when implementing Saiyan

on ASIC. The hardware cost of Saiyan, on the other hand, is around 27.2 USD, which can be also reduced sharply after ASIC fabrication.

5.3 Case Studies

Next, we run two real-world case studies to showcase packet re-transmission (§5.3.1) and frequency hopping (§5.3.2).

5.3.1 Packet re-transmission through the ACK mechanism

Setups. We integrate Saiyan into PLoRa and Aloba tags, which allows the tags to demodulate the feedback signals from the receiver and make an immediate packet re-transmission if needed. The link distance is set to 100 m.

Results. As shown in Figure 26, PLoRa and Aloba achieve 81.8% and 45.6% packet reception ratio (PRR) without packet re-transmission. The PRR of Aloba grows drastically from 45.6% to 70.1% when the Aloba tag is allowed to re-transmit the lost packet only once. The PRR then grows to 83.3$ and further to 95.5% when the Aloba tag re-transmits the lost packet twice and three times, respectively. The PRR of PLoRa shows the similar trend. These results demonstrate that Saiyan effectively improves the packet reception ratio for long-range LoRa backscatter systems.

5.3.2 Interference avoidance through channel hopping

As an ultra-low-power tag working on the ISM band, both PLoRa and Aloba are likely to bear strong in-band interference from other legacy RF devices working on the same band. We show that with Saiyan, these backscatter tags can demodulate the feedback signals from the receiver and switch to other channels to avoid interference.

Setups. We use PLoRa to demonstrate the feasibility of channel hopping. The PLoRa tag communicates with the receiver at the 434 MHz frequency band. It switches to the 434.5 MHz frequency band upon detecting the feedback signal from the receiver. We put a software-defined radio three meters away from the receiver to jam the channel at the 433 MHz frequency band.

Results. Figure 27 shows the CDF of PRR before and after the channel hopping. We can see the PRR is very low when the USRP jams the channel (dotted line). As the receiver

initiates a channel hopping command to the backscatter tag, we witness a significant lift on the PRR. In particular, the median PRR grows from 47% to 92% once PLoRa switches to another channel. This result clearly demonstrates that Saiyan can support better channel utilization through remote control.

6 Related Work

We review research topics relevant to Saiyan in this section.

RFID system. A passive RFID tag modulates sinusoidal tone from an RFID reader to transmit data [52, 58]. It can also demodulate amplitude-modulated (AM) signals from a nearby RFID reader [16, 26, 51, 53]. Specifically, the RFID tag downconverts the incident signal to the baseband and accumulates the signal power through an integrator circuit. Subsequently, it compares the accumulated power to a threshold to demodulate incident signals. Saiyan differs from passive RFID tags in two aspects. First, Saiyan demodulates frequency-modulated signal as opposed to amplitude-modulated signal. Second, Saiyan is designed for long-range backscatter systems whereas the passive RFID tag functions within only a few meters.

Ambient backscatter systems. Ambient backscatter systems empower backscatter tags to take the ambient wireless traffic as the carrier signals [14, 15, 18, 20, 23, 29–33, 35, 37, 39, 40, 47–50, 55–57, 60]. For example, WiFi backscatter [33] reuses WiFi signals as the carrier, thereby allowing for the backscatter tag to communicate with a commercial WiFi receiver. Interscatter [29] enables backscatter tags to modulate Bluetooth signals into WiFi signals. LoRa backscatter [47] allows backscatter tags to communicate over long distances by taking advantage of the noise resilience of LoRa symbols. These pioneer works have remarkably improved the throughput and the communication range of backscatter systems. Some recent works [37, 44, 55, 56, 59, 60] support a few types of downlink functionalities such as carrier sensing [37, 44, 55, 56, 59, 60] and packet detection [23, 40] at the packet level. For example, WiFi backscatter [33], Passive-WiFi [34], Interscatter [29], LoRa backscatter [47], and Netscatter [24] use the presence and absence of carrier packets to convey downlink data. However, they cannot demodulate downlink packets at the symbol level, particularly under long-range settings. Saiyan can serve as an important building block to the existing long-range backscatter systems, where the on-demand retransmission is needed due to the drastic packet loss.

Low-power demodulator. With the growth of low-power IoT market, the research community has shifted the focus to the design and implementation of low-power RF receivers, *e.g.*, by replacing the active components with their passive counterparts, or by offloading the power-intensive functions to external devices. Ensworth et al. [19] proposed a 2.4 GHz low-power BLE receiver that offloads the RF local oscillator to an external device. Carlos et al. [41] proposed a low-power 802.15.4 receiver that could demodulate phase-modulated

ZigBee signals at orders of magnitude lower power consumption compared with the standard 802.15.4 receiver. However, the working range of this low-power receiver is limited to tens of centimeters, which sets a strong barrier towards the practical deployment. Turbo charging [39] designs a multi-antenna cancellation circuit to facilitate the signal demodulation on backscatter tags. Similarly, full-duplex backscatter [38] enables a backscatter tag to demodulate the instantaneous feedback signal from another backscatter tag. Saiyan differs from these systems in two aspects. First, Saiyan is designed for demodulating frequency-modulated signals as opposed to phase or amplitude modulated signals. Second, Saiyan can support up to 180 m demodulation range, whereas all the aforementioned systems function within only tens of centimeters.

SAW filter. The SAW filter has been widely adopted by wireless communication systems such as telecommunications [25], radar [54], and aerospace communications [45], *etc.* These systems leverage the low-distortion and minimal passband variation of the SAW filter to filter out noise and interference signals. Furthermore, medical devices transform a SAW filter into a sensor for in-situ detection (*e.g.*, detecting chemical gas concentration) [21, 28]. Different from all the above applications, Saiyan exploits the sharp frequency response of the SAW filter to demodulate frequency-modulated signal.

7 Conclusion

We have presented the design, implementation, and evaluation of Saiyan, the first-of-its-kind low-power demodulator for LoRa backscatter systems. Saiyan allows LoRa backscatter tags to demodulate the command or feedback signals from a remote access point that is hundreds of meters away. With such capability, the backscatter tag can realize a plethora of networking functionalities, such as packet re-transmission, channel hopping, and rate adaptation. Field study shows that Saiyan outperforms state-of-the-art systems by 3.5–5× in terms of demodulation range. The ASIC simulation shows that the power consumption of Saiyan is around 93.2 μW.

Acknowledgment

References

[1] B39431-B3790-Z810 by Qualcomm-RF360 SAW filters. Webpage.

[2] 3 dBi omni-directional antenna in 433 *MHz*. Webpage.

[3] Energy harvesting chip LTC3105. Webpage.

[4] Introduction to SAW filter theory & design techniques. Webpage.

[5] LoRa Alliance. Webpage.

[6] LoRa receiver. Webpage.

[7] LoRa transceivers SX1276RF1JAS in 433 *MHz*. Webpage.

[8] Low-power amplifier transistors 2N222. Webpage.

[9] Low-power comparator NCS2202. Webpage.

[10] Realization of Differential Circuit. Webpage.

[11] Silicon oscillators LTC6907. Webpage.

[12] The RF in RFID. Webpage.

[13] Ultra-low power microcontroller Apollo2 Blue. Webpage.

[14] Mohamed R. Abdelhamid, Ruicong Chen, Joonhyuk Cho, Anantha P. Chandrakasan, and Fadel Adib. Self-reconfigurable micro-implants for cross-tissue wireless and batteryless connectivity. In *Proceedings of ACM MobiCom, Virtual event, September 21-25*, 2020.

[15] Dinesh Bharadia, Kiran Raj Joshi, Manikanta Kotaru, and Sachin Katti. BackFi: High throughput WiFi backscatter. In *Proceedings of ACM SIGCOMM, Budapest, Hungary, August 20-25*, 2018.

[16] Binbin Chen, Ziling Zhou, and Haifeng Yu. Understanding RFID counting protocols. In *Proceedings of ACM MobiCom, Miami, Florida, USA, September 20-October 4*, 2013.

[17] Chunyuan Chiu, Zhencheng Zhang, and Tsung Hsien Lin. Design of a 0.6-V, 429-MHz FSK transceiver using Q-enhanced and direct power transfer techniques in 90-nm CMOS. *IEEE Journal of Solid-State Circuit*, 55(1):3024–3035, 2020.

[18] Farzan Dehbashi, Ali Abedi, Tim Brecht, and Omid Abari. Verification: Can WiFi backscatter replace RFID? In *Proceedings of ACM MobiCom, New Orleans, USA, October 25-29*, 2021.

[19] Joshua F. Ensworth, Alexander T. Hoang, and Matthew S. Reynolds. A low power 2.4 GHz super-heterodyne receiver architecture with external LO for wirelessly powered backscatter tags and sensors. In *Proceedings of IEEE RFID, Phoenix, AZ, May 9-11*, 2017.

[20] Joshua F. Ensworth and Matthew S. Reynolds. Every smart phone is a backscatter reader: Modulated backscatter compatibility with Bluetooth 4.0 Low Energy (BLE) devices. In *Proceedings of IEEE RFID, San Diego, CA, USA, April 15-17*, 2015.

[21] Fahim, Mainuddin, U. Mittal, Jitender Kumar, A. T. Nimal, and M. U. Sharma. Single chip readout electronics for SAW based gas sensor systems. In *Proceedings of IEEE SENSORS, Glasgow, UK, October 29- November 1*, 2012.

[22] Amalinda Gamage, Jansen Christian Liando, Chaojie Gu, Tan Rui, and Mo Li. LMAC: Efficient carrier-sense multiple access for LoRa. In *Proceedings of ACM MobiCom, Virtual event, September 21-25*, 2020.

[23] Xiuzhen Guo, Longfei Shangguan, Yuan He, Jia Zhang, Haotian Jiang, Awais Ahmad Siddiqi, and Yunhao Liu. Aloba: Rethinking on-off keying modulation for ambient LoRa backscatter. In *Proceedings of ACM SenSys, Virtual event, November 16-19*, 2020.

[24] Mehrdad Hessar, Ali Najafi, and Shyamnath Gollakota. Netscatter: Enabling large-scale backscatter networks. In *Proceedings of USENIX NSDI, Santa Clara, CA, March 16-18*, 2016.

[25] Tzuhsuan Hsu, Fengchieh Su, Kuanju Tseng, and Minghuang Li. Low loss and wideband surface acoustic wave devices in thin film Lithium Niobate on Insulator (LNOI) platform. In *Proceedings of 34th International Conference on Micro Electro Mechanical Systems (MEMS), Gainesville, FL, USA, January 25-29*, 2021.

[26] Pan Hu, Pengyu Zhang, and Deepak Ganesan. Laissez-faire: Fully asymmetric backscatter communication. In *Proceedings of ACM SIGCOMM, London, United Kingdom, August 17-21*, 2015.

[27] Xiongchuan Huang, Guido Dolmans, Harmke de Groot, and John R. Long. Noise and sensitivity in RF envelope detection receivers. *IEEE Transactions on Circuit and Systems*, 60(10):1549–7747, 2013.

[28] Tarikul Islam, Upendra Mittal, A T Nimal, and M U Sharma. Surface Acoustic Wave (SAW) vapour sensor using 70 MHz SAW oscillator. In *Proceedings of 6th International Conference on Sensing Technology (ICST), Kolkata, India, December18-21*, 2012.

[29] Vikram Iyer, Vamsi Talla, Bryce Kellogg, Shyamnath Gollakota, and Joshua Smith. Inter-technology backscatter: Towards internet connectivity for implanted devices. In *Proceedings of ACM SIGCOMM, Salvador, Brazil, August 22-26*, 2016.

[30] Junsu Jang and Fadel Adib. Underwater backscatter networking. In *Proceedings of ACM SIGCOMM, Beijing, China, August 19-24*, 2019.

[31] Zhang Jianhui, Zheng Siwen, Zhang Tianhao, Wang Mengmeng, and Li Zhi. Charge-aware duty cycling methods for wireless systems under energy harvesting heterogeneity. *ACM Transactions on Sensor Networks*, 16(15):1–23, 2020.

[32] Mohamad Katanbaf, Anthony Weinand, and Vamsi Talla. Simplifying backscatter deployment: Full-duplex LoRa backscatter. In *Proceedings of USENIX NSDI, virtual, April 12-14*, 2021.

[33] Bryce Kellogg, Aaron Parks, Shyamnath Gollakota, Joshua R. Smith, and David Wetherall. WiFi backscatter: Internet connectivity for RF-powered devices. In *Proceedings of ACM SIGCOMM, Chicago, USA, August 17-22*, 2014.

[34] Bryce Kellogg, Vamsi Talla, Joshua R. Smith, and Shyamnath Gollakot. Passive WiFi: Bringing low power to WiFi transmissions. In *Proceedings of USENIX NSDI, Santa Clara, CA, March 16-18*, 2018.

[35] Songfan Li, Chong Zhang, Yihang Song, Hui Zheng, Lu Liu, Li Lu, and Mo Li. Internet-of-microchips: Direct radio-to-bus communication with SPI backscatter. In *Proceedings of ACM MobiCom, Virtual event, September 21-25*, 2020.

[36] Alexei N. Liashuk, Sergey A. Zavyalov, Aleksandr N. Lepetaev, Anatoliy V. Kosykh, and Igor V. Khomenko. Digitally temperature compensated SAW oscillator based on the new excitation circuit. In *Proceedings of IEEE International Frequency Control Symposium & the European Frequency and Time Forum, Denver, CO, USA, April 12-16*, 2015.

[37] Vincent Liu, Aaron Parks, Vamsi Talla, Shyamnath Gollakota, David Wetherall, and Joshua R. Smith. Ambient backscatter: Wireless communication out of thin air. In *Proceedings of ACM SIGCOMM, Hong Kong, China, August 12-16*, 2013.

[38] Vincent Liu, Vamsi Talla, and Shyamnath Gollakota. Enabling instantaneous feedback with full-duplex backscatter. In *Proceedings of ACM MobiCom, Maui, Hawaii, USA, September 7-11*, 2014.

[39] Aaron N. Parks, Angli Liu, Shyamnath Gollakota, and Joshua R. Smith. Turbocharging ambient backscatter communication. In *Proceedings of ACM SIGCOMM, Chicago, USA, August 17-22*, 2014.

[40] Yao Peng, Longfei Shangguan, Yue Hu, Yujie Qian, Xianshang Lin, Xiaojiang Chen, Dingyi Fang, and Kyle Jamieson. PLoRa: A passive long-range data network from ambient LoRa transmissions. In *Proceedings of ACM SIGCOMM, Budapest, Hungary, August 20-25*, 2018.

[41] Carlos Perez-Penichet, Claro Noda, Ambuj Varshney, and Thiemo Voigt. Battery-free 802.15.4 receiver. In *Proceedings of IEEE/ACM IPSN, Porto, Portugal, April 11-13*, 2018.

[42] Brecht Reynders, Franco Minucci, Erma Perenda, Hazem Sallouha, , and Roberto Calvo Palomino. Fast-settling feedforward automatic gain control based on a new gain control approach. *IEEE Transactions on Circuits and Systems*, 61(9):651–655, 2014.

[43] Brecht Reynders, Franco Minucci, Erma Perenda, Hazem Sallouha, Roberto Calvo, Yago Lizarribar, Markus Fuchs, Matthias Schafer, Markus Engel, Bertold Van den Bergh, Sofie Pollin, Domenico Giustiniano, Gerome Bovet, and Vincent Lenders. SkySense: Terrestrial and aerial spectrum use analysed using lightweight sensing technology with weather balloons. In *Proceedings of ACM MobiSys, Online, June 16-19*, 2020.

[44] Mohammad Rostami, Karthik Sundaresan, Eugene Chai, Sampath Rangarajan, and Deepak Ganesan. Redefining passive in backscattering with commodity devices. In *Proceedings of ACM MobiCom, Virtual event, September 21-25*, 2020.

[45] Franz Seifert, Helmut Stocker, and Otto Franz. The first SAW based IFF system and its operation in Austrian aerospace defence. In *Proceedings of IEEE History of Telecommunications Conference, Paris, France, eptember 11- 12*, 2008.

[46] Joshua R. Smith, Alanson P. Sample, Pauline S. Powledge, Sumit Roy, and Alexander V. Mamishev. A wirelessly-powered platform for sensing and computation. In *Proceedings of ACM UbiComp, Orange County, California, September 17-21*, 2006.

[47] Vamsi Talla, Mehrdad Hessar, Bryce Kellogg, Ali Najafi, Joshua R. Smith, and Shyamnath Gollakota. LoRa backscatter: Enabling the vision of ubiquitous connectivity. In *Proceedings of ACM UbiComp, Maui, HI, USA, September 11-15*, 2017.

[48] Ambuj Varshney and Lorenzo Corneo. Tunnel emitter: Tunnel diode based low-power carrier emitters for backscatter tags. In *Proceedings of ACM MobiCom, Virtual event, September 21-25*, 2020.

[49] Ambuj Varshney, Oliver Harms, Carlos Perez Penichet, Christian Rohner, and Thiemo Voigt Frederik Hermans. LoRea: A backscatter architecture that achieves a long communication range. In *Proceedings of ACM SenSys, Delft, Netherlands, November 06-08*, 2017.

[50] Anran Wang, Vikram Iyer, Vamsi Talla, Joshua R. Smith, and Shyamnath Gollakota. FM backscatter: Enabling connected cities and smart fabrics. In *Proceedings of USENIX NSDI, Boston, MA, USA, March 27-29*, 2017.

[51] Ju Wang, Liqiong Chang, Shourya Aggarwal, Omid Abari, and Srinivasan Keshav. Soil moisture sensing with commodity RFID systems. In *Proceedings of ACM MobiSys, Toronto, Ontario, Canada, June 16-19*, 2020.

[52] Jue Wang, Haitham Hassanieh, Dina Katabi, and Piotr Indyk. Efficient and reliable low-power backscatter networks. In *Proceedings of ACM SIGCOMM, Helsinki, Finland, August 13-17*, 2012.

[53] Davide Zanetti, Boris Danev, and Srdjan Apkun. Physical-layer identification of UHF RFID tags. In *Proceedings of ACM MobiCom, Chicago, Illinois, USA, September 20-24*, 2010.

[54] Peng Zhang, Houjun Wang, Li Li, Lianping Guo, and Ping Wang. FPGA based echo delay control method for pulse radar testing. In *Proceedings of 13th IEEE International Conference on Electronic Measurement and Instruments (ICEMI), Yangzhou, China, October 20-22*, 2017.

[55] Pengyu Zhang, Dinesh Bharadia, Kiran Joshi, and Sachin Katti. HitchHike: Practical backscatter using commodity WiFi. In *Proceedings of ACM SenSys, Stanford, CA, USA, November 14-16*, 2016.

[56] Pengyu Zhang, Colleen Josephson, Dinesh Bharadia, and Sachin Katti. FreeRider: Backscatter communication using commodity radios. In *Proceedings of ACM CONEXT, Incheon, Republic of Korea, December 12-15*, 2017.

[57] Pengyu Zhang, Mohammad Rostami, Pan Hu, and Deepak Ganesan. Enabling practical backscatter communication for on-body sensors. In *Proceedings of ACM SIGCOMM, Salvador, Brazil, August 22-26*, 2016.

[58] Yufan Zhang, Ertao Li, and Yihua Zhu. Energy-efficient Dual-codebook–based backscatter communications for wireless powered networks. *ACM Transactions on Sensor Networks*, 17(9):1–20, 2021.

[59] Jia Zhao, Wei Gong, and Jiangchuan Liu. Towards scalable backscatter sensor mesh with decodable relay and distributed excitation. In *Proceedings of ACM MobiSys, Virtual event, June 16-19*, 2020.

[60] Renjie Zhao, Fengyuan Zhu, Yuda Feng, Siyuan Peng, Xiaohua Tian, Hui Yu, and Xinbing Wang. OFDMA-enabled WiFi backscatter. In *Proceedings of ACM MobiCom, Los Cabos, Mexico, October 21-25*, 2019.

A Appendix

In this section, we prove the infeasibility of RLC resonant circuit to realize LoRa frequency-amplitude transformation.

A.1 The Infeasibility of RLC Resonant Circuit

The center frequency ω_0, the passband $\Delta\omega$, and the quality factor Q of a resonant circuit satisfy that:

$$Q = \frac{\omega_0}{\Delta\omega} \tag{6}$$

A higher Q value leads to a narrower passband width. Taking a step further, the quality factor Q is determined by the resistance R, inductance L, and capacitance C of this circuit following the equation:

$$Q = \sqrt{L}/(R \cdot \sqrt{C}) \tag{7}$$

Given a constant center frequency of $\omega_0 = 1/(2\pi\sqrt{LC})$, we can deduce the capacitance C satisfy that:

$$C = \frac{1}{Q\omega_0 R} = \frac{\Delta\omega}{\omega_0^2 R} \tag{8}$$

Generally, the equivalent R of RF circuit is 50 Ω. Taking LoRa signals working on 433 MHz frequency band (with 500 KHz bandwidth) as an example, this requires C to be as low as $5.2 \times 10^{-14} pF$.

Graham: Synchronizing Clocks by
Leveraging Local Clock Properties

Ali Najafi
Meta[†]

Michael Wei
VMware Research

[†]*Work done while at VMware*

Abstract

High performance, strongly consistent applications are beginning to require scalable sub-microsecond clock synchronization. State-of-the-art clock synchronization focuses on improving accuracy or frequency of synchronization, ignoring the properties of the local clock: lost of connectivity to the remote clock means synchronization failure.

Our system, Graham, leverages the fact that the local clock still keeps time even when connectivity is lost and builds a failure model using the characteristics of the local clock and the desired synchronization accuracy. Graham characterizes the local clock using commodity sensors present in nearly every server and leverages this data to further improve clock accuracy, increasing the tolerance of Graham to failures. Graham reduces the clock drift of a commodity server by up to 2000×, reducing the maximum assumed drift in most situations from 200ppm to 100ppb.

1 Introduction

The ever increasing performance demands of strongly consistent distributed applications has driven a desire for tightly synchronized clocks. Instead of communicating over the network, servers can establish an order over messages using a timestamp from a local clock [7, 18, 30]. Leveraging synchronized clocks has become more pervasive as applications require tighter latencies that approach the latency of the network itself. However, deploying finely synchronized clocks at scale remains a significant challenge, often requiring the use of specialized hardware [20, 22],.

In an ideal system, synchronizing clocks would be a trivial task. Clocks would never *drift* (lose or gain time) and a synchronized clock would stay synchronized forever. In real systems, however, clocks drift, so synchronization needs to be done frequently to keep clocks in time. Spanner [7], for example, assumes 200ppm drift, which translates into 200μs/s, a second roughly every hour, or a minute every 4 days. This drift increases clock uncertainty (ε) and limits the performance of

applications leveraging clocks, which must wait out the uncertainty. State-of-the-art systems today assume high clock drift and focus on increasing synchronization precision and frequency, with specialized hardware performing as many as 10K synchronizations per second to achieve sub-microsecond clock synchronization [17, 20, 22, 28]. Furthermore, systems assume missed synchronizations result in loss of synchronization, resulting in potentially unnecessary shutdowns due to clock uncertainty exceeding application requirements [20].

The clocks which drive the processor and timestamping hardware, however, are required to drift far less than these systems expect: datasheets from several vendors specify a clock crystal with at least ±20ppm temperature stability [9, 26]. An unstable clock could cause the system to violate the tight timing requirements required by the processor, memory and I/O subsystem. Local clocks can be much more stable than most systems assume. If we know that a system has lower drift, we can reduce the rate of synchronization, tolerate synchornization failures, reduce network congestion and the overhead of processing synchronization messages, as well as avoid the use of specialized hardware [4].

In this paper, we describe Graham[1], a system which models the stability of the local clock to determine the required synchronization rate. Graham leverages sensors available in every commodity server to characterize the clock against an accurate reference clock, such as GPS, PTP or even NTP. Graham uses this characterization to build a synchronization model, which determines how frequently the system must be synchronized and how many synchronization failures can be tolerated, and can achieve below 1ppm drift. In the servers we tested, we were able to achieve 100ppb stability in most cases, which is over 2000× better than the max drift rate assumed by Spanner. The guiding principle behind Graham is to improve the clock in software without adding additional hardware. This approach is challenging because existing sensors are not designed to characterize clocks, and are located at varying

---

[1]Named after George Graham (1673–1751), a clockmaker who improved the pendulum clock's accuracy by compensating for changes in pendulum length due to temperature.

distances away from the oscillators that drive system clocks. To address this challenge, Graham characterizes the system by observing the effect of temperature fluctuations at various sensors on clock error between synchronizations.

The contributions of this paper are:

- We debunk the myth that commodity computers have unstable clocks.

- We describe how to automatically characterize computer clocks using commodity sensors.

- We show that this characterization can be used to greatly reduce synchronization rates, resulting in 100ppb stability without specialized hardware.

2 Clock Generation and Synchronization

The term *clock* is often used for several related concepts. In this paper, we will use *clock* to mean a counter which is incremented at some frequency and can be used to measure time. Clocks are driven by *clock signals*, which oscillate between low and high logical states. A clock driven by this signal increments on a *clock edge*, which is the transition between two logical states (typically, the rising edge is used). Clock signals are provided by *clock sources*, which are typically quartz crystal oscillators in modern computers. In this section, we provide background on how clocks are generated and synchronized in a typical computer system.

2.1 A typical Linux Intel x86 clock system

Clock systems are architecture and vendor dependent, so we focus on a typical Intel Linux x86 machine as a model clock system. An Intel x86 system consists of multiple clocks which are driven by multiple clock sources. Some of the clocks accessible to users include the timestamp counter (TSC), real-time clock (RTC) and the precision time protocol clock (PTP). Each of these clocks run at different frequencies and serves different purposes, and which clock software ultimately can access has been shown to vary [23].

For the purposes of this paper, we center on the TSC, the clock typically accessed by applications via `clock_gettime(2)`. This clock is driven by a clock signal known as BCLK (typically 100MHz). The BCLK is driven by a phase-locked loop (PLL) which multiplies the frequency of a quartz crystal (48MHz on C620 ICC [9]). The BCLK is an important signal which not only drives the logic in the processor, but the memory controller and other components, depending on the processor model. Adjusting the BCLK is often done when overclocking by changing PLL parameters, but large adjustments can result in system instability and lockup.

So far, we have described how Linux enables applications to read a clock. In order to be able to compare one clock to another, clocks must be synchronized. Most Linux distributions rely on `ntpd` [24] or `chrony` [5] to synchronize local clocks to a remote server with a reference clock synchronized to wall clock time (UTC) via a time source such as GPS using the NTP protocol. The NTP protocol estimates the network delay between the server to client by dividing the round-trip delay in half and can achieve on the order of 1ms-100ms time synchronization error, with error increasing as the delay becomes more asymmetrical. In addition, since NTP is run in software, synchronization is subject to software jitter such as scheduling and interrupt handling which prevents NTP accuracy below 0.5ms, even in ideal conditions.

To achieve sub-microsecond accuracy, PTP (IEEE 1588) reduces software jitter [10]. First, instead of acting as a service where clients request the time, a PTP server continuously broadcasts the current time at periodic intervals. Clients estimate the network delay by sending a special message to the server to compute the round trip time and dividing that time in half. Finally, PTP introduces a new hardware clock located on the network card itself. This clock is driven by a different quartz crystal at the network card, usually corresponding to the frequency needed to drive the card's transceivers (25MHz for 10Gb Ethernet). The network card can capture the synchronization packets as they arrive to synchronize the PTP clock to the server, eliminating the inaccuracy introduced by software jitter. In Linux, `phc2sys` synchronizes the TSC clock to the PTP clock.

The accuracy of PTP is dependent on accurate delay estimation. Recognizing this limitation, Huygens [14] and Tick Tock [6] use coded probes and support vector machines to filter out queued packets from round trip delay estimation. Some commercial PTP implementations use packet delay variation (PDV) filters [27], and compensate for known latencies in the receive and transmit paths.

Because of clock drift, synchronization frequency is also important. While most of the latency sensitive paths of PTP are in hardware, it is still software driven, limiting the frequency of PTP synchronization, especially when filters are used that necessarily discard some synchronization data. This can be problematic if clock synchronization requirements are tight and clock drift is high. For instance, Huygens has a default sync interval of 2 seconds. A clock with 200ppm (0.02%) of drift will accumulate up to $400\mu s$ of drift between missed synchronizations. If there is a single transient synchronization failure resulting in a 4 second interval, up to $800\mu s$ of drift would accumulate, which would be problematic if an application required sub-microsecond clock accuracy. To increase synchronization frequency, most solutions require specialized hardware. For example, DTP modifies the Ethernet physical layer to exchange messages at the frequency of microseconds while reducing network delay nondeterminism [17]. Sundial leverages specialized hardware that synchronizes every $100\mu s$ and performs fast failure detection to notify software to recover by finding a backup clock [20].

2.2 Holdover Time

Notably, current state-of-the-art systems do not attempt to characterize the *holdover time* of the clock, which refers to the amount of time a clock can remain accurate without a synchronization. For instance, Spanner and Sundial both assume a static 200ppm maximum drift. If the maximum time uncertainty bound (ε) is 1μs, then a clock with 200ppm drift (200μs/s) will only be able to holdover the clock for 5ms without synchronization before *potentially* exceeding ε. The formula for holdover time can be given as:

$$t_h = \frac{\varepsilon}{\mathrm{d}f} \qquad (1)$$

Where t_h is the holdover time, ε is the maximum time uncertainty, and $\mathrm{d}f$ is the clock drift. 200ppm, however, is very conservative: most quartz crystals used for computer systems are specified on the order of 100ppm of error, and only when operated under extreme operating conditions. In the next section, we describe how we can characterize oscillator error, and use this characterization to increase the holdover time.

2.3 Characterizing Oscillator Error

Oscillators provide the clock signals that ultimately drive the clocks used in computer systems, and are the source of most clock error. The most common oscillator in use in nearly all computers today is a quartz crystal, which uses the pizeoelectric properties of quartz to produce a clock signal at a given frequency.

Quartz is cut to resonate at a given frequency, however, as the cut is a mechanical process, tolerances in the cut process may result in a resonant frequency which is slightly offset from the advertised frequency, known as the frequency tolerance. Since any error in the cut is usually fixed, this tolerance results in a fixed offset from the advertised frequency. In typical computer crystals, this error is usually in the 50ppm range. Lower tolerances require more accurate (e.g., fine laser) cuts and are significantly more expensive.

Quartz crystals also age over time as mechanical devices which are constantly vibrating, slowly deviating from their advertised frequency. This error is usually small (5ppm/year) [1], and also results in a slight frequency offset.

So far, we have described sources of quartz crystal oscillator error which are relatively constant. As physical devices, the frequency of quartz crystals are also affected by environmental changes. The most prominent factor is temperature [36], which can result in a significant change in frequency over the crystal's operating temperature range. While temperature can induce variations in the frequency of the crystal, the temperature-frequency response of crystals are quite deterministic: in fact, some crystal manufacturers produce the response curve on the crystal datasheet. Typical crystals produce anywhere from a 30ppm-100ppm change in frequency over their operating temperature ranges [1].

Table 1: Frequency Error in Standard Quartz Oscillators

Name	Typical Range	Typical
Tolerance	$\pm$50 ppm	-
Aging	$\pm$5 ppm/year	-
Temperature	$\pm$1 ppm/$^\circ$C	$\pm$15 ppm (25-40°C)
Voltage	$\pm$1 ppm/V	$\pm$0.1 ppm ($\pm$2% 3.3V)
Load	$\pm$0.1 ppm/pF	$\pm$0.1 ppm ($\pm$10% 15pF)
Acceleration	0.1 ppb/G	0 @ Rest
Time Dilation	0.1 ppq/m	0 @ Sea Level

In addition to temperature, a variety of other environmental factors will affect the frequency of the oscillator. However, these factors contribute a relatively small amount of frequency error compared to temperature. Changes in supply voltage usually result in a 0.1ppm-5ppb change in frequency. Another factor is variation in the load capacitance: in order for the crystal to resonate at the expected frequency, the correct amount of capacitance is required. Since the capacitors used to provide the load capacitance also have tolerances, the capacitance can vary depending on the properties of the capacitors used. Typically, load capacitance error is specified at 0.1ppm-5ppb [29, 33]. The frequency of quartz crystals are also sensitive to acceleration, depending on the axis it is applied to. For ordinary quartz crystals, this is typically in the range of 0.1-10ppb/G [29, 33]. For a 500G shock, such as that specified in MIL-STD-883H, representative of a device dropping to the floor, frequency error could be as high as 1ppm [19, 33]. Note that the recommendation for operational vibration and shock limits in datacenters is less than 5G [16] which is well below 500G. Finally, crystals are even sensitive to relativity: a crystal closer to the gravitational field of the earth will have a lower frequency than a crystal further away, such as on a mountain or in space. This error is around 0.1ppq/m from sea level, or ≈ 0.9ppt at the top of Mount Everest or ≈ 3ppb from geostationary orbit [33].

These sources of error are a result of the physical properties of quartz, and the data collected in Table 1 are collected from the datasheets of various quartz oscillators used in servers [1, 19, 29, 33, 36].

2.4 Debunking the Myth of Unstable Clocks

As we have seen, most of the frequency error in a quartz oscillator is either relatively static or dependent on temperature. Voltage and load only contribute a small amount of error and should be within small tolerances (otherwise, other parts of the system may begin failing). Servers in most datacenters are stationary, so the effects of acceleration and time dilation should be constant.

Static error can be easily corrected if it can be learned: if we learn that our crystal resonates at 32.769 KHz instead of 32.768 KHz, we simply need to adjust our accounting of time,

perhaps by using 32769 as a divider instead of 2^{15}. If our synchronization error is minimal and we keep the temperature constant, we can learn this value over several synchronization passes. NTPd and chrony both try to learn the static drift using the *driftfile*.

Most state-of-the-art systems, however, combine static and dynamic error in their uncertainty calculations, resulting in the assumption of an unstable clock. For instance, Sundial assumes that the clock error of their oscillator is 100ppm, but this number includes the static tolerance error from the cut, which is easily learned. Moreover, even if they had chosen a ±100ppm temperature tolerance crystal, this shift would be over the entire operating range, as in a shift from -30°C to 85°C. An overheating server moving from 60°C to 80°C would experience only about 20ppm change in drift from temperature, an order of magnitude less than the conservative 200ppm error used in spanner.

In practice, most crystals used to generate processor clocks have temperature tolerances in the range of ±20ppm. Intel Chipset Integrated Clock Controllers (ICC), for example, specify "Total of crystal cut accuracy, frequency variations due to temperature, parasitics, load capacitance variations is recommended to be less than 90ppm" [9], and external clock generators such as the common CK420BQ used in Intel systems specify a cut tolerance of ±20ppm and a temperature tolerance of ±20ppm over the entire operating range [26]. If we can filter out the static error, we will be left with 20ppm temperature error. Then this clock will have a 1 μs holdover time of 50ms, a 10× improvement over the 200ppm assumption.

2.5 Software Temperature Compensation

Once we have corrected the static frequency error, temperature remains as the dominant source of frequency error. This effect is well known, and software compensation techniques are described in the literature [13, 15, 25]. In computers, chrony can correct for temperature errors given the temperature-frequency relationship and a temperature sensor. In wireless networks, where minimizing clock error is critical, environment and temperature aware compensation are used [34, 35].

While temperature-frequency curves are sometimes published on the datasheet of a crystal, using them to correct errors on a commodity computer system requires knowing the crystal used. This can be difficult even for an expert given the small markings on most crystal packages. Moreover, the crystal used can be different even across the same model of motherboards, since manufacturers may substitute functionally equivalent parts due to cost or supply-chain reasons. Unless the system was purpose built with temperature correction in mind, temperature sensors are likely located some distance away from the crystal. Therefore, selecting the right temperature sensor may be a challenge. However, correcting for temperature error can effectively reduce the frequency

error of the crystal to less than 1ppm, resulting in a 1 μs holdover time of 1s, a 200 × improvement over Spanner's assumption.

2.6 Other Oscillators

Many applications outside of general-purpose computing, such as wireless require low frequency error over a wide temperature range. The temperature compensated crystal oscillator (TCXO) consists of quartz crystal with a temperature compensation circuit and reduces the effect of temperature to ≈ ±1ppm of error. The oven compensated crystal oscillator (OCXO) takes temperature control one step further and places the crystal in a miniature oven which keeps the crystal at a constant temperature, reducing temperature effects to ≈ ±1ppb. This oven can be doubled (DOCXO) to achieve ≈ ±0.1ppb of temperature error. Atomic oscillators, which work based on electron transitions, can provide even more stability: rubidium oscillators provide up to 0.0002ppb/s. The cost of these oscillators is often cited as prohibitive, but can be quite inexpensive, relative to specialized hardware. For instance a 48MHz TCXO at 0.5ppm suitable for driving an Intel ICC costs around USD $2 [11], and a 25MHz OCXO at 10ppb suitable for driving a CK420BQ clock synthesizer costs around USD $70 [2].

While replacing the oscillators in computer systems might be an option in new, future hardware, it is an invasive and expensive procedure for existing hardware. The focus of Graham is to democratize accurate clocks using only existing hardware. Using software techniques, we can achieve low error without adding additional hardware.

3 Clocks and Sensors In Servers

In order to understand how temperature sensors can be used to estimate clock error in commodity systems, we studied the sensor and time configuration of a variety of platforms. One unexpected challenge was the difficulty of accurately measuring clock error.

Clocks. The Linux pulse-per-second (PPS) [21] facility provides a mechanism for delivering an accurate reference time. PPS devices are devices that accurately emit a low-jitter pulse every second. A PPS driver calls the `pps_event` API whenever the pulse is received, and the kernel records the timestamp associated with that pulse. Typically, this pulse is a signal that causes an interrupt, and the PPS API is called by an interrupt service routine (ISR). However, even when using very low jitter PPS devices, such as the ublox ZED-F9T [32] GPS timing module that advertises ±4 ns jitter, we saw jitter over 10μs. As we diagnosed the problem, we saw several sources of jitter throughout the hardware and software stack which made it difficult for our driver to call `pps_event` in a timely manner after the pulse interrupt is raised.

Our initial approach was to use GPS dongles with PPS support over USB[2], which are inexpensive (USD ≈$10), readily available, and usable on nearly every server. The GPS device presents itself as a serial device, and the PPS interrupt is encapsulated as a message over the USB bus. We saw that the polling message-driven nature of USB resulted in high jitter: not only was there a ≈100μs delay (which is easily corrected for), but also ±10μs of jitter that made it difficult to accurately time the pulse. Our next attempt involved using a FPGA to deliver an interrupt over PCIe, since PCIe slots are readily available in most commodity servers. However, while PCIe offered less jitter PCIe interrupts are also message signaled and also saw as much as ±5μs of jitter dependent on device traffic and serial transceiver jitter.

We needed a low-latency interrupt pin to accurately capture the PPS signal. We ended up resorting to using the legacy serial port, which exposes interrupts pins on the device carrier detect (DCD) and clear to send (CTS) lines. Unlike PCIe and USB, these legacy ports drive an interrupt pin on the low-pin count (LPC) bus and offer much lower jitter, on the order of 1μs. Even with the serial port, we still saw significant "blips" in our PPS signal. To reduce those blips, we made several changes: first, we pinned the serial port interrupt to a single core, disabled power management, disabled all watchdogs, installed a "lowlatency" kernel, turned on interrupt threading and set the serial interrupt priority to realtime. While these changes reduced the number of blips, there was still periodic noise present which made time daemons such as chrony detect as much as 10ppm of drift change over a second. This drift only disappeared when we forced the C-state of the machine to C0, disabling idling. This surprised us: the CPU advertised `FEATURE_NONSTOP_TSC`, so the TSC should not be affected by C-States. We realized that the most likely scenario was that when idling was enabled, the CPU would take a non-deterministic amount of time to wake up from sleep and fire the ISR that eventually causes `pps_event` to be recorded.

To deal with this scenario, we took advantage of the two time pulse outputs of the ZED-F9T module and connected the second time pulse to the CTS serial line. We configured the second time pulse with a 400ns delay from the first one, and modified the kernel PPS serial line discipline driver to only record the second pulse if is 400ns ± 100ns from the first pulse. While this caused some pulses to disappear, it greatly reduced the jitter we observed. To compensate for lost time pulses, we changed the time pulse frequency from 1Hz to 3Hz. Removing this software jitter enabled us to see that the clock was actually fairly stable over long periods of time, only deviating by about .5ppm per hour, as seen in Figure 1. We suspected most of this deviation was due to the rising ambient temperature.

[2]To expose the PPS signal, we used a common FT232H USB-to-RS232 converter and connected the PPS line to the DCD signal expected by the PPS serial line discipline driver.
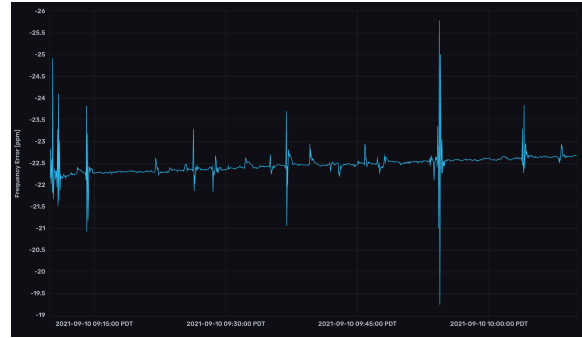
(a) Frequency Error Without Dual Time Pulse, all C-states enabled.
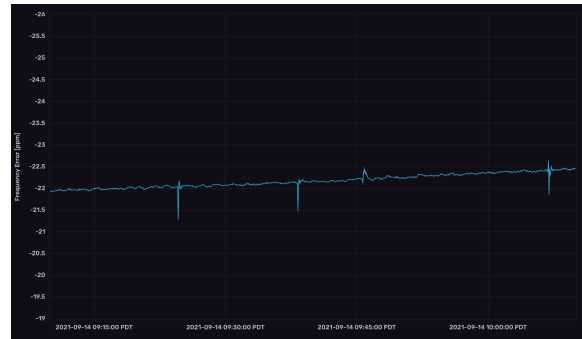
(b) Frequency Error with Dual Time Pulse

Figure 1: **Software Frequency Error.** Interrupts and system activity give the illusion clock error.

Table 2: Systems Evaluated and Temperature Sensors

Name	Type	Crystal Location	Sensors
Server	2S 1U Rack	Near Chipset	50
Workstation	Desktop	Chipset	8
Pi 4	SoC	Under SoC	1
Pi 3	SoC	Under SoC	1

Sensors. Modern computer systems are littered with sensors for environmental conditions. The original use of these sensors were to monitor alarm conditions: for example, to shut off the system if there are abnormally high temperatures that would cause instability, or if a voltage regulator malfunctions. A more recent use of temperature sensors is for thermal throttling, which reduces the frequency of a processor or GPU based on the temperature. The goal of Graham is to reuse these temperature sensors for the purpose of performing software-based temperature compensation.

Using these temperature sensors can be challenging because their location relative to the clock crystal is not consistent. While crystals are usually located near the clock generator, the clock generator can be located in a number of locations, which might not be at all near a temperature sensor. Systems also have a varying number of sensors, as shown in Table 2. The server platform we evaluated, for example,
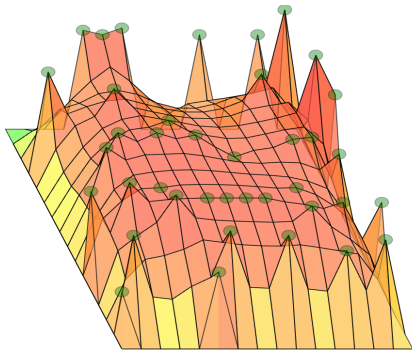
Figure 2: **Server Platform Temperature Map.** The server platform contains over 50 sensors with approximate positions labeled.
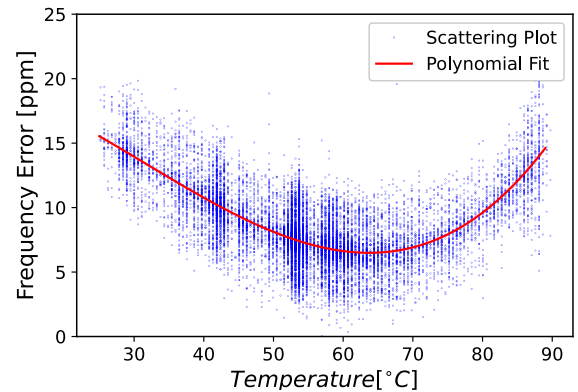

Figure 3: **Temperatures and Polynomial Fit.** Even though there is variation in the measured delay, a polynomial fit curve can still be plotted against it.

has nearly 50 sensors (Figure 2). However, even though the platform provides the position of these sensors, it is still of little help to determine which sensor is closest to the crystal. As an additional challenge, not all temperatures offer the same precision. For instance, some of the sensors in the server platform only reported $\pm 10°C$ changes, likely because they were designed only for use as an alarm. Finally, the response time of the sensors may vary depending on various environmental factors. For instance, a sensor located near the large copper ground plane of the motherboard may respond slower to rising temperatures than a sensor located on a the thinner PCB of a DIMM. An ideal sensor has high precision and responds quickly to changes in the same way as the crystal.

Establishing the Ground Truth. Armed with an accurate timing signal and a number of candidate sensors, our next goal is to attempt to establish the "ground truth", or the temperature-clock error response curve. If we can determine the clock error given a certain temperature, then we can correct the clock even in the absence of the accurate timing signal.

Nearly all quartz crystals used in computers today are AT-cut crystals. Their frequency relationship with temperature can be described by a 3rd order equation [3, 8, 12, 36]:

$$\Delta f_T = k_0 + k_1 T + k_2 T^2 + k_3 T^3 \qquad (2)$$

where Δf_T is the crystal frequency error due to temperature, T is the crystal temperature and k_i are coefficients of the frequency versus temperature curve. To find the relationship of the clock frequency versus temperature we need to solve for the k_i parameters using synchronization messages from a reference clock. Unfortunately, since the sensor data is noisy, we may need to obtain many temperature points to "average out" the sensor error. This required designing an experiment which required many passes, and was difficult to perform on a server platform. As a result, we performed most of our ground truth tests on the Raspberry Pi (Pi 3/Pi 4) SoC systems, though we

show our full implementation of Graham in action on desktop and server platforms in Section 5. While the Raspberry Pi is an ARM-based SoC, it runs Linux like the x86 system and has a clock driven by a quartz crystal on the underside of the SoC PCB.

The Pi, as a bare SoC system, allowed us to easily subject it to various temperatures. The Pi includes a temperature monitor which measures the core SoC temperature. We provided an accurate PPS timing pulse using a uBlox Neo-M8N GPS module [31] to a Pi GPIO and exposed it to various temperatures using either a hair dryer or ice bucket. We used the difference in timing ticks between PPS signals to calculate the estimated frequency error of the crystal, and the result is plotted in Figure 3. The distribution we saw was around ± 5 ppm and probably attributable to interrupt delay and sensor error.

Once we saw that we were able to capture the temperature-error relationship, we wanted to ensure that the data we were generating was repeatable, so we collected several traces using varying temperature patterns, all exercising the same temperature range. Figure 4 shows that the curve we generated was similar even with different temperature inputs.

Next, we wanted to see if the curves differed across devices. Figure 5 shows that even across devices of the same model, curves are significantly different. Even the same crystal model could be cut slightly differently, resulting in two 25MHz crystals which are for example, 24.997MHz and 25.001MHz that meet the tolerance requirement, but yield different curves.

Finally, because age can have an effect on the crystals, we wanted to test if we could observe a change in the curve with age. In Figure 6, we ran two tests with a 7 month time difference, obtaining two slightly different curves, as expected. The 1ppm offset we obtained roughly matches the aging expected by a regular quartz crystal during this time period.

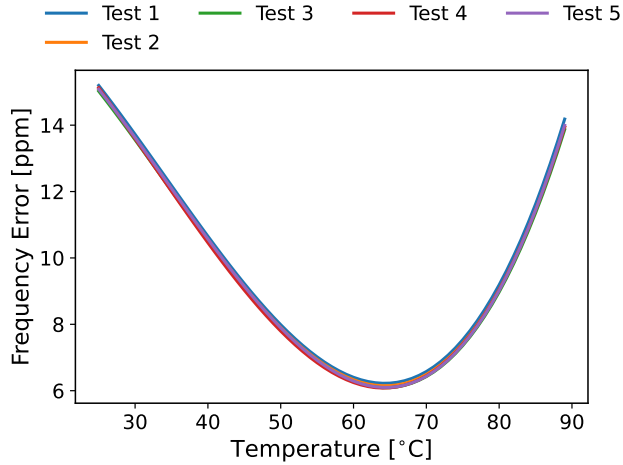Now that we have obtained the ground truth using an accu-

Figure 4: **Repeatability of the Curve.** We measured several traces using different temperature patterns (Test 1-5) by varying the use of ice and the hair dryer and we obtained similar temperature curves.

rate PPS signal, we use this knowledge to guide us in scaling our solution to many devices. Since each device will have its own unique curve, it became clear to us that we needed to design a way to automatically learn the curve of each device.

4 Graham Design

The overall approach of Graham is to learn the temperature-clock error relationship by fitting curves as new data points are learned. Unlike the experiments we designed when trying to learn the ground truth, we cannot expect to be able to point a hair dryer or dump a production server in ice. In addition, since a truly scalable solution should not require a precise PPS timing signal, we need to ensure that we can perform this learning with traditional synchronization protocols such as NTP or PTP. As a result, Graham must fit these curves over time on incomplete and noisy data. Once we determine that the we have observed enough data points, we can use the derived curve to correct the time error. To fit this data on a curve, we begin by formalizing the variables and equations required to solve for the time error.

4.1 Formulating the problem

We previously described the relationship of the crystal error with temperature as a cubic polynomial in Equation 2. However, we cannot directly measure the frequency of the crystal to obtain the error. Instead, we can obtain two timestamps from the clock using a known time interval and calculate the difference to see how much it deviates from the expected difference.
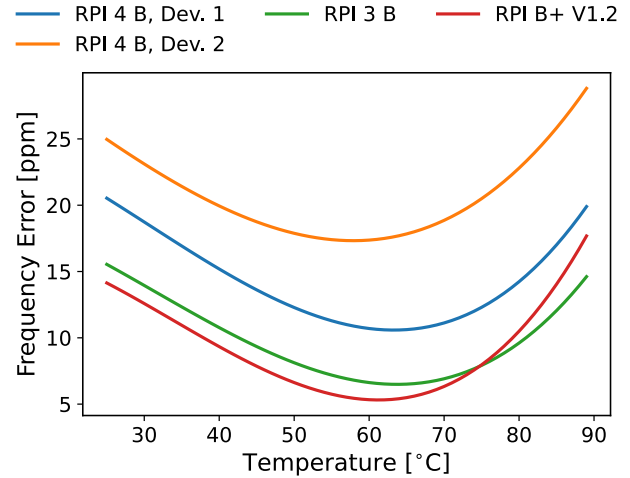
Figure 5: **Curves Across Devices.** We observed that different devices, even of the same model had varying curves.
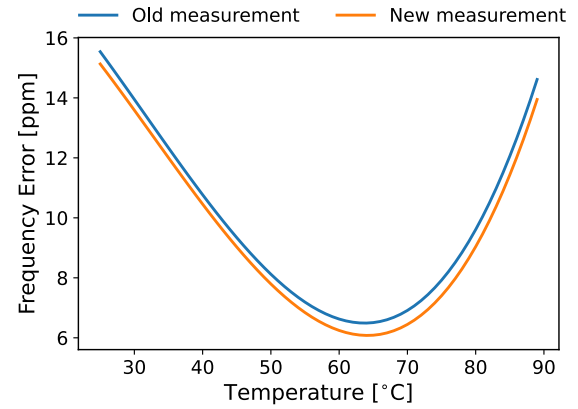
Figure 6: **Aging of Devices.** As a device ages, the curve can change due to crystal aging effects.

For example, a clock crystal may have an ideal frequency (f_0) of 32.768KHz. We would expect two timestamps taken exactly 1 second apart to have a difference of 1 (Δts_i). But if we actually observe 1.5 seconds (Δts_o), then we know the actual frequency is 49.152KHz (f_1), or $1.5 \times f_0$. If we subtract the two frequencies, we obtain 16.384KHz of frequency error (Δf). We can express this as an equation:

$$\Delta f \Delta ts_i = \Delta ts_o - \Delta ts_i \tag{3}$$

in which Δf is the relative frequency error. If we assume most of the frequency error is from temperature, we can replace Δf_T in Equation 2 with Δf. Then we obtain:

$$(k_0 + k_1 T + k_2 T^2 + k_3 T^3)\Delta ts_i = \Delta ts_o - \Delta ts_i \tag{4}$$

Eq. 4 is a linear equation with 4 unknowns – k_0, k_1, k_2 and k_3. Timestamp interval Δts_o can be obtained from the

system's local clock and the timestamp interval Δts_i can be obtained from synchronization messages. If we receive N synchronization messages then we can build N linear equations as follows:

$$AK = B \qquad (5)$$

in which A, K and B are matrices equal to:

$$K = \begin{bmatrix} k_0 \\ k_1 \\ k_2 \\ k_3 \end{bmatrix}, \quad A = \begin{bmatrix} 1 & T_1 & T_1^2 & T_1^3 \\ 1 & T_2 & T_2^2 & T_2^3 \\ \dots & \dots & \dots & \dots \\ 1 & T_N & T_N^2 & T_N^3 \end{bmatrix} \qquad (6)$$

$$B = \begin{bmatrix} \Delta ts_{o,1} - \Delta ts_{i,1} \\ \Delta ts_{o,2} - \Delta ts_{i,2} \\ \dots \\ \Delta ts_{o,N} - \Delta t_{i,N} \end{bmatrix} \qquad (7)$$

in which T_N, $\Delta ts_{i,N}$ and $\Delta t_{o,N}$ are respective parameters for the Nth synchronization message and equation. Graham solves Eq. 5 using linear least square methods.

So far, we assumed that the temperature is constant for the duration of Δts_o. If the synchronization messages are infrequent, as in the case of a protocol such as NTP, the temperature can change during this period. To solve this problem, Graham records temperatures during this period and when it receives a synchronization message, it aggregates the effects of temperatures. Assume there are n intervals in which we record temperatures during a period. The equation for the jth time interval is:

$$\Delta f_j \Delta ts_{i,j} = \Delta t_{o,j} - \Delta t_{i,j} \qquad (8)$$

$$\Delta t_o = \sum_j^n \Delta t_{o,j} \qquad (9)$$

$$\Delta t_i = \sum_j^n \Delta t_{i,j} \qquad (10)$$

$$\sum_j^n \Delta f_j \Delta t_{i,j} = \sum_j^n \Delta t_{o,j} - \sum_j^n \Delta t_{i,j} \qquad (11)$$

Using Eq. 2, 9 and 10, we get:

$$k_0 \sum_j^n \Delta ts_{i,j} + k_1 \sum_j^n T_j \Delta ts_{i,j} + k_2 \sum_j^n T_j^2 \Delta ts_{i,j} \\ + k_3 \sum_j^n T_j^3 \Delta ts_{i,j} = \Delta ts_o - \Delta ts_i \qquad (12)$$

where T_j is the temperature at the jth time interval. Note that, $\Delta ts_{i,j}$ is an unknown parameter. We can be approximated it by $\alpha \Delta ts_{o,j}$ in which $\alpha = \frac{\Delta t_i}{\Delta t_o}$.

$$k_0 \sum_j^n \Delta ts_{o,j} + k_1 \sum_j^n T_j \Delta ts_{o,j} \\ + k_2 \sum_j^n T_j^2 \Delta ts_{o,j} + k_3 \sum_j^n T_j^3 \Delta ts_{o,j} = \\ \frac{\Delta ts_o - \Delta ts_i}{\alpha} \qquad (13)$$

Similar to Eq. 4, Eq. 13 is a linear equation with 4 unknowns and we can solve it using similar linear least square methods.

4.2 Implementation

We implemented a prototype daemon in C which solves for the equations by using temperature sensors exposed through sysfs or a network management interface such as SNMP. We record temperatures with $1°C$ precision at a configurable frequency, which defaults to 1Hz. For synchronization data, we modified chrony to collect the Δts_o and Δts_i necessary from synchronization messages over NTP.

Graham keeps a FIFO queue of equations with known size for each temperature, bounding the number of equations that need to be solved. Graham assumes an operating temperature range of 40-80°C and does not start applying corrections until the curve errors are within 20ppm. Graham constantly collects temperature data to learn the curve before corrections are applied.

4.3 Addressing practical issues

In 4.1, we assumed an ideal case in which all the known parameters to solve for the clock frequency versus temperature are accurate. However, that is not the case in practical systems. We outline these inaccuracies and non-idealities and explain how we can address them.

4.3.1 Timestamp Error

There are two main sources of timing error in the system:

Error in $\Delta t_{o,i}$. Since the temperature changes happen in the timescale of seconds, even several milliseconds error in the observed $\Delta t_{o,i}$ values will have a limited effect on the result.

Error in $(\Delta t_i - \Delta t_o)$. This value is the combination of 3 parameters: crystal frequency error (Δf), jitter in timestamps and network asymmetry from the time server to our system. Graham is interested in only Δf, but the last two parameters are error (δt_{err}) and add noise to our measurements.

$$\Delta t_o - \Delta t_i = \Delta f \Delta t_i + \delta t_{err} \qquad (14)$$

Note that δt_{err} is only dependent on the type of timestamping (software and hardware) and the method of the synchronization (NTP, PTP, PPS and ...). The error in curve estimation

is determined by $\frac{\delta t_{err}}{\Delta t_i}$. Therefore, as we increase Δt_i, the first term in Eq. 14 increases while the second term is constant and we can increase the curve estimation accuracy. Moreover, δt_{err} can be modeled as a random variable with zero mean. As we increase the number of equations, we can average out the δt_{err} and in turn the lower the estimation error. By having a high enough number of equations and building equations for longer durations we can increase the curve estimation accuracy.

4.3.2 Temperature Sensor Challenges

Leveraging already existing temperature sensors requires addressing several challenges:

Accuracy. Temperature sensor accuracy has limited effect on correction performance since both learning the relationship of the clock frequency versus temperature and applying the correction is done using the same temperature sensor.

Precision. Low precision means that temperature measurement readings have a random variability. Having a higher number of equations will average out these random errors. This means that as the temperature sensor's statistical measurement variance increase we need higher number of equations.

Responsiveness. A temperature sensor which does not respond to temperature in the same way the crystal does will limit the effectiveness and potentially contribute to error. This responsiveness of a sensor can be measured by checking the temperature error curve. In a system with multiple temperature-error curves, we select the sensor which minimizes the frequency error during learning runs.

4.3.3 Computation Accuracy

The computed curve is only accurate for the temperature ranges that the system has experienced. For example, if Graham only has equations for temperatures from $50°C$ to $80°C$, the curve is accurate in that range and close to boundaries of that range. As we go far from this boundary the accuracy of the curve decreases. One of the main reasons for this is that the temperature-error curve is cubic, but the typical operating range of the server is only within a small convex region of the curve. Two of the roots are likely at the extreme temperature ranges, and one root is likely in the extreme negative (below freezing region).

To exercise a variety of temperature ranges without using a heater or ice, we load the CPU and allow the system to cool off.

5 Evaluation

Our evaluation of Graham is motivated by the following:

- How effective is learning over a noisy synchronization channel such as NTP? (§5.2)

(a) After 40 Eq ($\approx$12h) (b) After 80 Eq ($\approx$24h)

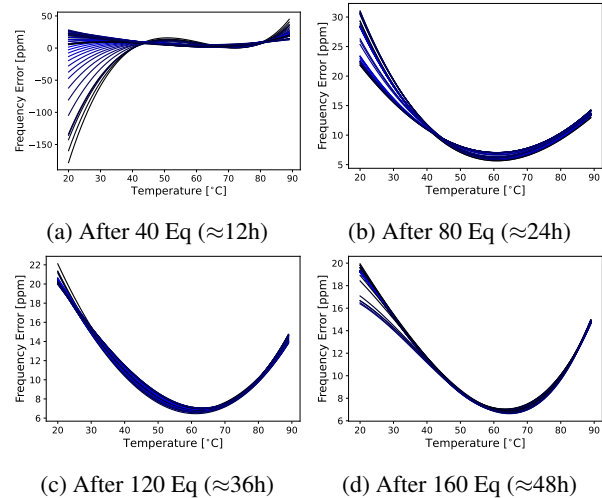(c) After 120 Eq ($\approx$36h) (d) After 160 Eq ($\approx$48h)

Figure 7: **NTP Learning.** While learning over NTP takes longer, the curve converges towards the same curve produced by faster, more accurate synchronization sources.

- What is the holdover time Graham can achieve, and how many synchronization failures can it tolerate? (§5.3)

- Can Graham compensate for rapid changes in temperature, as in with a HVAC failure?(§5.4)

Test Platforms. The primary system requirement to be able to apply Grahamis the presence of a temperature sensor which is present in nearly all modern computer systems. We evaluated Graham on several platforms, as shown in Table 2. For the Pi tests, we used a ublox M8N [31] GPS receiver with a time pulse accuracy of $\pm$60ns (99%). The M8N module does not specify jitter, but we observed $\pm$20ns jitter using a RIGOL MSO5074 oscilloscope. For the x86-based platforms, we used a ublox ZED-F9T [32] GPS module which specifies a time pulse accuracy of $\pm$5ns (1σ), and a jitter of $\pm$4ns. In our tests, we are mainly concerned about jitter, as the timing accuracy specifies the accuracy of the timing pulse to GPS time, and these GPS modules have their own TCXO oscillator.

5.1 Learning over PPS

We obtained baseline curves with Graham using PPS. With PPS, we generate 1 new equation per second, corresponding to the frequency of the synchronization signal. As shown in Figure 8, even though the temperature data we used to generate each curve was quite different, the curves are almost the same. While the curves look similar, the constants for each curve varies. This is because there are many cubic equations which can fit the small convex portion of the curve that we observe within the operating temperature range.
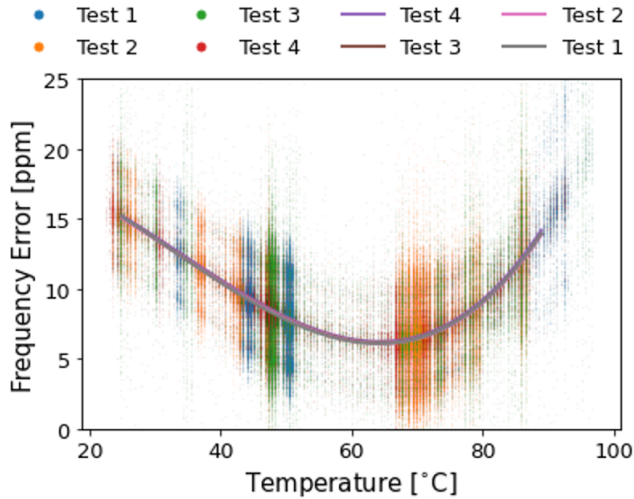
Figure 8: **PPS Curves and Input Data.** Curves learned over each experiment have a similar shape despite having variable input data.
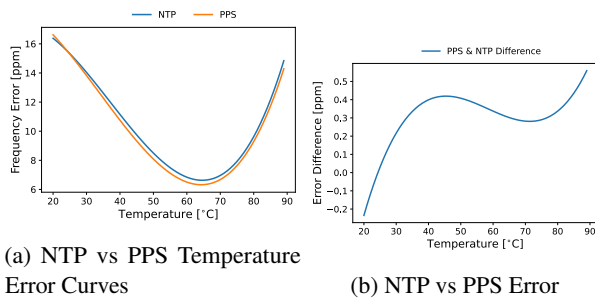

(a) NTP vs PPS Temperature Error Curves

(b) NTP vs PPS Error

Figure 9: **NTP Performance Compared to PPS.** The learned curves for NTP are within 0.5ppm of the PPS curves for most of the operating range.

5.2 Learning over NTP

To evaluate learning over NTP, we used chrony to obtain NTP synchronization data for Graham against public NTP servers over a standard home cable broadband connection, with a ping latency to the NTP server between 30–40ms. This resulted in synchronization accuracy in the ms range. In order to compensate for this, we needed to use high $\Delta t s_o$. For NTP, we use $\Delta t s_o$ = 1000s, which results in one equation every 1000s as opposed to 1 equation per second with PPS. Note that $\Delta t s_o$ is independent of the synchronization periods and intervals used by chrony, which has its own algorithm for NTP synchronization frequency.

Figure 7 shows the 160 equations we collected over the course of a 48 hour run. This resulted in a curve within ±0.5ppm of the curve generated using PPS signals, as shown in Figure 9. We suspect that the error of the curve is not constant because of lack of data points at temperature extremes

for both sets of data.

5.3 Holdover

Once we have learned the temperature-error relationship, we wanted to evaluate how well Graham's time frequency correction would perform in the absence of synchronization messages. To test the accuracy of the frequency correction, we recorded the accurate PPS time pulse, but did not provide it to Graham. We measured the accuracy of Graham's time correction versus the real time. We then exposed the system to a new temperature trace.

Pi Experiments. Figure 10 shows a trace of one of these experiments on the Pi 3. In this particular experiment we exposed the system to both ambient air effects of the 8 hour time period as well as artificial cooling (ice) and heating (hair dryer). The red vertical line shows the rapid growth of time error if Graham did not perform any compensation. At 620s, this well exceeds $5000\mu s$ of drift, which corresponds to the the 8ppm of temperature drift Graham is trying to correct for. On the other hand, Graham's corrections perform very well, never exceeding $1500\mu s$ of error over the course of the entire 8 hour run, even though the temperature is shifting significantly. For most of the test, the slope never exceeds 100ns/s of error, which means the clock is performing as well as one with only 100ppb of error, a $200\times$ improvement over the performance of the 20ppm crystal, performing nearly as well as a high quality TCXO or some OCXOs. We can calculate the holdover time using the slope from Equation 15, given a maximum time uncertainty (ε). If ε=1μs, then the holdover time during the 100ns/s region is:

$$t_h = \frac{1\mu s}{100ns/s} \qquad (15)$$

or $t_h = 10s$. In other words, the corrected clock will not exceed 1 μs of error for at least 10 seconds *without* any additional synchronization. This would enable more infrequent synchronizations, or enable the system to tolerate the very real potential of missed synchronization messages. In one part of the graph, we experience a 330ns/s slope, when the temperature exceeds 85°C. We speculate that this slope is because the training temperature data we used had very few points at or above this temperature. 330ppb still is very good: we obtain a 1μs holdover time of 3 seconds, which still allows for lower frequency synchronizations.

Server Experiments. We also evaluated the holdover time on desktop and server x86 systems. These systems are much more complex and contain multiple sensors and fans, so Graham needs to determine which sensor works best, given a variety of factors. There are also multiple components which can generate heat load, which vary from system to system. Notably, the many fans in the server made it more difficult to
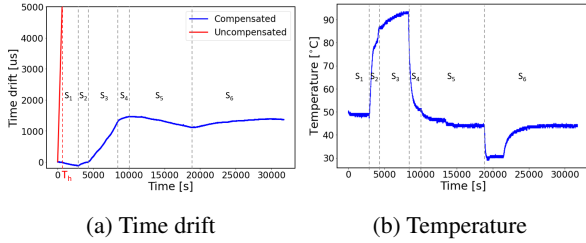
(a) Time drift

(b) Temperature

Figure 10: **Holdover.** The uncompensated temperature drift quickly increases while Graham is able to maintain the time with minimal drift. The slope of each part of the graph corresponds to the frequency error performance: $s_1 = 50$, $s_2 = 80$, $s_3 = 330$, $s_4 = 100$, $s_5 = 30$, $s_6 = 15$ ns/s.
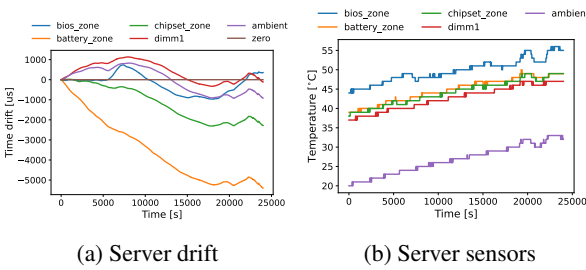

(a) Server drift

(b) Server sensors

Figure 11: **Warming Server Holdover.** With a server sitting in a garage on a hot summer day, Graham is able to maintain 0.1ppm of error.

create rapid changes in temperature. To get a picture of the server sensor's performance, we performed a 24 hour learning run exposing the server to various temperatures while heating it from the fan intake and letting it cool via ambient cooling, and running `stress-ng` in various modes to create load on the system. We then exposed the server to a new temperature curve.

Figure 11 shows the holdover graph for the server during one of our first tests, which is just ambient warming of the server in a garage on a hot summer day. We selected the 5 best performing sensors. Surprisingly, even though we thought the "chipset_zone sensor" would perform the best, "dimm1" actually produced the best correction curve. We wanted to ensure that this would be the case even in a loaded system, so we performed a memory test using `stress-ng` to see if heat from a memory load would affect our learned result. Figure 12 shows the holdover curves from that run, with the memory test running at time 0. The DIMM 1 sensor still remained one of the top performing sensors, producing less than 200μs of drift over the first 2000 seconds of the run, or 0.1ppm error. Many of the other sensors perform well too, likely because they experience similar patterns of temperature changes. The impact of the load, however, can be seen across Figure 11 and Figure 12: while the ambient temperature works well without a load, its performance is worse when a load is present. In all our runs with the server workload, we never observed more
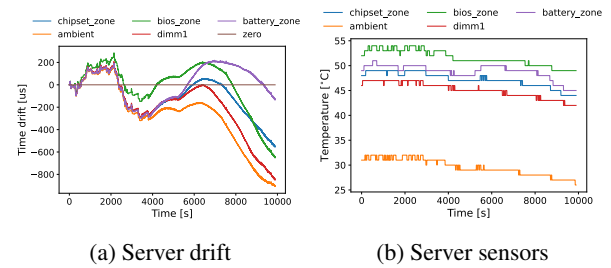

(a) Server drift

(b) Server sensors

Figure 12: **Memory Load Holdover.** The DIMM sensor remains the best sensor, even when the server is under memory load.

than a 0.2ppm error with Graham.

Desktop Experiments. Finally, we evaluated Graham on a typical desktop machine. Unlike the server, which is fully instrumented with sensors throughout, the desktop machine we used only had a few sensors exposed by default, just on the CPU die and the DIMMs. However, during our experiments, we made an error to include the output of the fan sensors (in RPM) as training data. Surprisingly, the fan sensors worked well even though they were not directly measuring the temperature. We suspect that the speed of the fan is driven by a combination of the ambient temperature, (which is not exposed to the user) and dynamic CPU load by the hardware monitor. However, we ended up using the second core sensor, which is located closer to the chipset and crystal. This gave us 0.1ppm error on nearly all experiments.

Figure 13 shows a peculiar experiment on the desktop platform where we failed to expose the server to all temperature points. In the first 2000 seconds we run a CPU load experiment, which resulted in the a higher than expected error (0.5ppm vs 0.1ppm). After debugging, we realized this was because the temperatures we exposed to Graham during testing (Figure 13c) were not learned (Figure 13d). In particular, the testing temperatures were above 70°C for the first 2000s, while the learning temperatures were below. Still, we thought this test showed that even without learning temperatures, Graham can provide some correction to the temperature error.

5.4 Rapid Changes

One of the often cited sources of timing instability in computer systems is a thermal shock event, such as an HVAC failure. To evaluate Graham's performance in dealing with a rapid thermal shock, we used the Pi system and pointed a hair dryer directly at it, attempting to raise the temperature rapidly to the maximum operating temperature. As with the holdover tests, we turned off synchronization and only relied on Graham's temperature-based frequency error correction.

Figure 14 shows the time drift after correction by Graham (left) which results from the rapidly rising slope in tem-
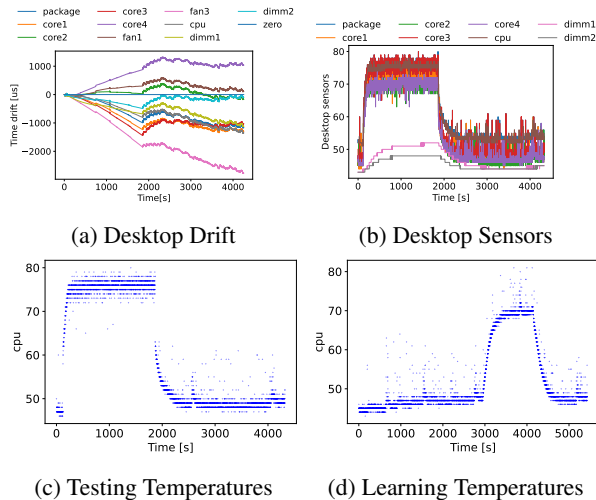
(a) Desktop Drift

(b) Desktop Sensors

(c) Testing Temperatures

(d) Learning Temperatures

Figure 13: **Desktop Holdover with Missing Points.** Even with missing points, Graham is able to make corrections to keep error within 0.5ppm.

(a) Time drift

(b) Temperature

(c) Uncorrected

Figure 14: **Thermal shock.** Even with a hair dryer's rapid heat, Graham is able to quickly compensate for errors in time, never drifting beyond 10μs.

perature (right) from the hair dryer. Using the hairdryer, we were able to produce a 2°C/s slope, which cools at about 0.2°C/s. While we feel that such fast heating is unlikely to happen, it may be representative of an HVAC failure, and is an indication of the robustness of Graham's temperature correction: the time drift never exceeds more than 10μs over the initial 25 second slope, a drift of only -0.4ppm. Once the temperature slope decreases, Graham is able to maintain the time without exceeding the initial 10μs of error. Without Graham, the system accumulates nearly 1ms of error during this time period (bottom).

6 Discussion

Our evaluation has shown that Graham can maintain clock frequency error below 1ppm using commodity sensors in a variety of conditions. Graham is only one part of the solution, however – while Graham can maintain a long holdover time, the synchronization maintained will only be as good as the initial synchronization.

Graham works in synergy with other synchronization mechanisms, such as Huygens [14], PTP [10] and FaRMv2 [28] to maintain synchronization. Our experiments with NTP show that Graham can maintain 1μs ϵ for 10 seconds after loss of synchronization. As Sundial [20] shows, however, missed synchronizations can occur for a number of reasons. For Huygens, significant CPU load on the system could occur causing the SVM processing to be delayed, and in PTP and FaRMv2, synchronization messages could be missed, leading to increased uncertainty of time. Using our 1μs holdover result for Graham, we could reduce the standard 1s synchronization frequency of PTP to 3s and tolerate 2 lost synchornization messages.

Graham also aims to democratize precise time by enabling commodity servers, desktops and even SoCs to have access to stable clocks without adding specialized hardware. One of the barriers we see in adopting precise time for these devices is the myth of the unstable clock, which is perpetuated by the challenge of measuring the drift in the clock in the first place. Software noise can give the illusion that a clock is drifting rapidly, even though hardware clocks are relatively stable. Unfortunately, without specialized hardware, drift is measured by software itself, further exacerbating the problem. By characterizing the clock. Graham enables applications to trust the hardware instead of relying on noisy software measurements.

In the future, we may consider incorporating multiple sensors to the equations Graham solves for better accuracy. As more applications require precise time, we expect systems with TCXOs or OCXOs to come on the market, and expect that Graham performs favorably against them.

7 Conclusion

It has been long thought that computer clocks are unstable, and that stability cannot be achieved without frequent synchronizations. We hope that this work dispels that myth and convinces the reader that much perceived clock instability is due to software measurement error. By understanding the sources of clock error, we have built Graham, which can reduce local clock error well below 1ppm using commodity clock sensors. Combined with an accurate synchronization source, Graham can maintain microsecond clock accuracy without additional hardware.

References

[1] Abracon. Abracon ABLKJO crystal oscillator. https://abracon.com/Precisiontiming/ABLJO.pdf.

[2] abracon. Aoc2012vajc ocxo datasheet. https://abracon.com/datasheets/AOC2012-Series.pdf.

[3] Abracon. Tuning fork crystals and oscillator. https://abracon.com/Support/Tuning-Fork-Crystals-and-Oscillator.pdf.

[4] BCM53903. Bcm53903 timing over packet (top) processor for precision timing applications. https://www.broadcom.com/products/embedded-and-networking-processors/communications/bcm53903.

[5] Chrony. chrony – introduction. https://chrony.tuxfamily.org/.

[6] Clockwork. Tick tock networks is now clockwork. https://www.clockwork.io.

[7] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22, 2013.

[8] CTS Corporation. Crystal basics. https://www.ctscorp.com/wp-content/uploads/Appnote-Crystal-Basics.pdf.

[9] Intel Corporation. C620 platform controller hub datasheet. https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/c620-series-chipset-datasheet.pdf.

[10] John C Eidson, Mike Fischer, and Joe White. Ieee-1588™ standard for a precision clock synchronization protocol for networked measurement and control systems. In *Proceedings of the 34th Annual Precise Time and Time Interval Systems and Applications Meeting*, pages 243–254, 2002.

[11] Epson. Tg2016smn ((tcxo / vc-tcxo) high stability). https://www5.epsondevice.com/en/products/tcxo/tg2016smn.html.

[12] Fil-Tech. Frequency-temperature behavior of at-cut crystals. https://www.filtech.com/tech-library/document/frequency-temperature-curve-cut-crystals/?dl=1.

[13] Marvin Frerking. *Crystal oscillator design and temperature compensation*. Springer Science & Business Media, 2012.

[14] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 81–94, 2018.

[15] D Habic and D Vasiljevic. Temperature compensation of crystal oscillators using microcontroller-/spl mu/ctcxo. In *Proceedings of IEEE 48th Annual Symposium on Frequency Control*, pages 587–593. IEEE, 1994.

[16] IBM. https://www.ibm.com/docs/en/power-blade-server/version_undefined?topic=planning-vibration-shock, Accessed: Feb 7 2022.

[17] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. Globally synchronized time via datacenter networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 454–467, 2016.

[18] Bojie Li, Gefei Zuo, Wei Bai, and Lintao Zhang. 1pipe: scalable total order communication in data center networks. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 78–92, 2021.

[19] Chao Li and Arvind Sridhar. Vibration and shock sensitivity: A comparative study of oscillators. *Texas Instruments, Dallas, TX, USA, Appl. Note SNAA296*, pages 1–11, 2017.

[20] Yuliang Li, Gautam Kumar, Hema Hariharan, Hassan Wassel, Peter Hochschild, Dave Platt, Simon Sabato, Minlan Yu, Nandita Dukkipati, Prashant Chandra, et al. Sundial: Fault-tolerant clock synchronization for datacenters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1171–1186, 2020.

[21] LinuxPPS. Linuxpps wiki. https://www.linuxpps.org.

[22] Pedro Moreira, Javier Serrano, Tomasz Wlostowski, Patrick Loschmidt, and Georg Gaderer. White rabbit: Sub-nanosecond timing distribution over ethernet. In *2009 International Symposium on Precision Clock Synchronization for Measurement, Control and Communication*, pages 1–5. IEEE, 2009.

[23] Ali Najafi, Amy Tai, and Michael Wei. Systems research is running out of time. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 65–71, 2021.

[24] ntpd. ntpd - network time protocol (ntp) daemon. `https://docs.ntpsec.org/latest/ntpd.html`.

[25] Bruce M Penrod. Adaptive temperature compensation of gps disciplined quartz and rubidium oscillators. In *Proceedings of 1996 IEEE International Frequency Control Symposium*, pages 980–987. IEEE, 1996.

[26] Renesas. 932sql456 - low-power ck420bq derivative for pcie separate clock architectures. `https://www.renesas.com/us/en/document/dst/932sql456-datasheet?r=166281`.

[27] Renesas. Limiting IEEE 1588 slave clock wander caused by packet delay variation. `https://www.renesas.com/us/en/document/whp/limiting-ieee-1588-slave-clock-wander-caused-packet-delay-variation`.

[28] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojević, Dushyanth Narayanan, and Miguel Castro. Fast general distributed transactions with opacity. In *Proceedings of the 2019 International Conference on Management of Data*, pages 433–448, 2019.

[29] SiTime. Sitime 5146 super-tcxo. `https://www.sitime.com/support/resource-library/datasheets/sit5146-datasheet`.

[30] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan Van-Benschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1493–1509, 2020.

[31] ublox. ublox neo-m8n datasheet. `https://www.u-blox.com/sites/default/files/NEO-M8-FW3_DataSheet_UBX-15031086.pdf`.

[32] ublox. ublox zed-f9t datasheet. `https://www.u-blox.com/en/docs/UBX-18053713`.

[33] John R Vig. Quartz crystal resonators and oscillators for frequency control and timing applications. a tutorial. *Nasa Sti/recon Technical Report N*, 95:19519, 1994.

[34] Miao Xu, Wenyuan Xu, Tingrui Han, and Zhiyun Lin. Energy-efficient time synchronization in wireless sensor networks via temperature-aware compensation. *ACM Transactions on Sensor Networks (TOSN)*, 12(2):1–29, 2016.

[35] Zhe Yang, Lin Cai, Yu Liu, and Jianping Pan. Environment-aware clock skew estimation and synchronization for wireless sensor networks. In *2012 Proceedings IEEE INFOCOM*, pages 1017–1025. IEEE, 2012.

[36] Hui Zhou, Charles Nicholls, Thomas Kunz, and Howard Schwartz. Frequency accuracy & stability dependencies of crystal oscillators. *Carleton University, Systems and Computer Engineering, Technical Report SCE-08-12*, 2008.

IA-CCF: Individual Accountability for Permissioned Ledgers

Alex Shamis[1,2], Peter Pietzuch[1,2], Burcu Canakci[*3], Miguel Castro[1], Cédric Fournet[1],
Edward Ashton[1], Amaury Chamayou[1], Sylvan Clebsch[1], Antoine Delignat-Lavaud[1], Matthew Kerner[4],
Julien Maffre[1], Olga Vrousgou[1], Christoph M. Wintersteiger[1], Manuel Costa[1], and Mark Russinovich[4]

[1]Microsoft Research, [2]Imperial College London, [3]Cornell University, [4]Microsoft Azure

Abstract

Permissioned ledger systems allow a consortium of members that do not trust one another to execute transactions safely on a set of replicas. Such systems typically use Byzantine fault tolerance (BFT) protocols to distribute trust, which only ensures safety when fewer than 1/3 of the replicas misbehave. Providing guarantees beyond this threshold is a challenge: current systems assume that the ledger is corrupt and fail to identify misbehaving replicas or hold the members that operate them accountable—instead all members share the blame.

We describe IA-CCF, a new permissioned ledger system that provides *individual accountability*. It can assign blame to the individual members that operate misbehaving replicas regardless of the number of misbehaving replicas or members. IA-CCF achieves this by signing and logging BFT protocol messages in the ledger, and by using Merkle trees to provide clients with succinct, universally-verifiable *receipts* as evidence of successful transaction execution. Anyone can *audit* the ledger against a set of receipts to discover inconsistencies and identify replicas that signed contradictory statements. IA-CCF also supports *changes* to consortium membership and replicas by tracking signing keys using a sub-ledger of governance transactions. IA-CCF provides strong disincentives to misbehavior with low overhead: it executes 47,000 tx/s while providing clients with receipts in two network round trips.

1 Introduction

Permissioned ledger systems, such as Hyperledger Fabric [4], Quorum [52] and Diem [3], allow a consortium of members that do not trust one another to deploy a trustworthy service on a set of replicas that they operate. These systems typically use protocols for Byzantine fault tolerant (BFT) state machine replication [12, 17, 20, 25, 37, 62] to distribute trust: clients send requests to execute transactions [59, 60] that are executed in a consistent order by the replicas. The results are recorded in a persistent, replicated ledger.

BFT protocols ensure safety (linearizability [29]) and liveness, but they can only do this if fewer than 1/3 of N replicas misbehave. With more misbehaving replicas, current permissioned ledger systems can no longer be trusted. When safety violations are detected, the whole service is deemed to have failed, and all members and replicas share the blame.

Current systems try to avoid this problem by increasing replication [25, 36, 62] or hardening individual replicas [54]. Adding replicas does not help if they are controlled by the same consortium members and thus do not behave independently. Increasing the number of consortium members, however, is challenging or even infeasible in practice. For example, the Diem Association [6] had 26 members, which prevented it from offering a service with more than 26 independent replicas; other consortia are smaller, which results in fewer independent replicas [7, 34, 50]. Even for large consortia with reputable companies, a persistent attacker may slowly compromise $N/3$ replicas over time, e.g., by exploiting lax security practices, bribing members' employees or exploiting software vulnerabilities. Without accountability after a service compromise, there is also no perceived reputational loss that would incentivize members to prevent or disclose these incidents [16, 24, 30].

The Confidential Consortium Framework (CCF) [54] uses trusted hardware [21, 35] to isolate replicas from operators and members, and it provides receipts that commit transaction execution to its ledger. However, CCF does not offer safety or individual accountability if the trusted hardware is compromised.

Prior work explores accountability for various types of distributed systems [1, 26, 27, 38, 64]. PeerReview [27] makes general message passing systems accountable. As we show in §6, applying such a general approach to a permissioned ledger system incurs high overhead: all messages must be signed, and auditing is expensive, because it correlates logs across many replicas. More recent work [14, 19, 53, 56] investigates accountability in BFT protocols and blockchains. These proposals, however, offer no guarantees when 2/3 or more replicas misbehave, because misbehaving replicas may rewrite the ledger history without detection.

We describe *Individual Accountability for CCF* (IA-CCF),

---

a BFT permissioned ledger system that identifies misbehaving replicas and assigns blame to the individual members that operate them, *even if all replicas misbehave*. Individual accountability provides strong disincentives for misbehavior.

IA-CCF is a prototype that extends CCF [54] with support for BFT and individual accountability, while retaining the same user programming model, key-value store, transaction execution engine, and model of governance for changes to the consortium membership and replica set.

IA-CCF supports individual accountability by introducing *Ledger PBFT* (L-PBFT), a new BFT state machine replication protocol that stores ordered transactions in the ledger together with the protocol messages from replicas that justify the execution order. L-PBFT maintains Merkle trees [42] over the ledger, and includes the roots of the trees in protocol messages. Since protocol messages are signed by the replicas, this commits them to the entire contents of the ledger.

IA-CCF then issues *receipts* to clients that provide succinct, universally-verifiable evidence that a transaction executed at a given position in the ledger. Receipts include signed protocol messages from multiple replicas that executed the transaction, thus binding them to a prefix of the ledger.

Given a collection of receipts that violates linearizability, anyone can audit the ledger against the receipts to assign blame to at least $N/3$ replicas. Auditing produces an irrefutable *universal proof-of-misbehavior* (uPoM) in the form of contradictory statements signed by the same replica. The uPoM can be used by an *enforcer*, e.g., a court, to punish the members responsible for the misbehaving replicas. To provide accountability when all replicas misbehave, the enforcer may have to compel members to produce a ledger, imposing sanctions otherwise. While this formally adds a weak synchrony assumption, the enforcer chooses a conservative timeout to make blaming correct members unlikely in practice.

As an example of auditing, a client Alice may have a receipt for a transaction that executed at index i in the ledger and deposited $1 million into client Bob's account. If Bob obtains the receipt from Alice and another receipt for a balance query transaction executed at index j ($j > i$) that does not show the balance, he may conduct an audit: he engages an enforcer to obtain the relevant ledger fragment, and replays the transactions between i and j to check for consistency with his receipts. If Bob is right, auditing produces a uPoM for at least $N/3$ replicas, which Bob sends to the enforcer to punish the consortium members responsible for the replicas.

To support changes to the consortium membership, IA-CCF uses *governance transactions* that alter the set of replicas and consortium members [54]. Governance transactions complicate receipt verification and auditing because they change the signing keys that must be considered. IA-CCF therefore records governance transactions in the ledger, which allows clients, replicas, and auditors to determine the set of valid signing keys. Clients do not need to keep the full ledger, but only receipts of governance transactions. Since
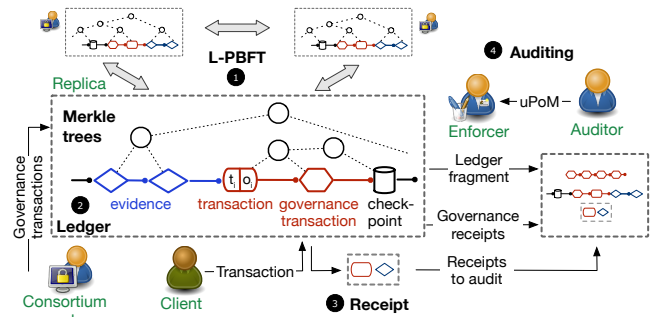

Fig. 1: IA-CCF permissioned ledger system

governance transactions are relatively rare, this *governance sub-ledger* is significantly smaller than the full ledger.

Our IA-CCF prototype provides individual accountability without compromising on throughput or latency: it implements a commitment scheme for transaction batches with only a single signature per replica. This enables clients to receive results with receipts after only two network round-trips. Our evaluation shows that IA-CCF can execute over 47,000 tx/s with low latency.

The contributions of IA-CCF and the paper structure are:

1. L-PBFT, a BFT state machine replication protocol that orders and stores transactions together with the protocol messages justifying the execution order in a ledger (§3.1, §3.2);
2. universally-verifiable client receipts that are generated efficiently with the ledger (§3.3);
3. an efficient auditing approach using the ledger and associated checkpoints, which produces short proofs-of-misbehavior (§4); and
4. a governance mechanism for changing members and replica sets, allowing auditing to assign blame even after members have left (§5).

2 Overview of IA-CCF

Fig. 1 shows IA-CCF's design. An IA-CCF deployment provides a *service*, with a well known name, to *clients*, which are identified by their signing keys. Clients send requests to execute *transactions* by calling stored procedures that define the service logic. Transactions are executed by *replicas* against a strictly-serializable *key-value store* that supports roll-back at transaction granularity. A transaction *request t* reads and/or writes multiple key-value pairs and produces a transaction *result o*.

Consortium *members*, also identified by their signing keys, own the service. They may be added or removed over the service lifetime. For this, members issue *governance transactions*, which change the consortium membership, add or remove replicas, and update stored procedures. The first governance transaction, the *genesis transaction gt*, defines the initial members and replicas. Its hash is the service name.

❶ **Ledger PBFT (L-PBFT)** is a BFT state machine replication protocol used by replicas to order transactions. L-PBFT is based on PBFT [17]. It provides linearizability

and liveness if at most $f = \lceil N/3 \rceil - 1$ out of N replicas fail in a partially-synchronous environment [23].

❷ **Ledger.** L-PBFT maintains an append-only *ledger*, which stores each transaction request t and result o at a ledger index i. Since the consortium membership and the replica set are dynamic, the ledger also records governance transactions. They form a *governance sub-ledger*, which can be used to learn the public signing keys of active replicas and members at any index i.

To assign blame, the ledger also includes *evidence* that a transaction batch was committed by a quorum of replicas. This evidence consists of at least $N-f$ signed L-PBFT protocol messages for a batch. Finally, the ledger stores periodic checkpoints of the key-value store, allowing its state to be reconstructed by replaying the ledger from a checkpoint cp.

All entries in the ledger are bound by Merkle trees. Protocol messages for a transaction batch contain the roots of the Merkle trees. This commits replicas to the whole ledger while allowing succinct existence proofs for entries.

❸ **Receipts** are created by replicas and returned to clients. They bind request execution to members via the replicas' signatures over Merkle tree roots that contains the executed request and the ledger's history. If two or more receipts are inconsistent with any linearizable execution, at least $f+1$ replicas must have signed contradictory statements and can thus be assigned blame.

More precisely, a receipt R for $\langle t, i, o \rangle$ states that request t was executed at index i and produced result o. The receipt consists of $N-f$ protocol messages for t's batch, signed by different replicas, and a path from a Merkle tree root to the leaf that contains an entry for $\langle t, i, o \rangle$.

Clients may obtain receipts from a reply to a request they sent, from replicas, or from other clients. To validate a receipt, clients must check its signatures using the signing keys determined by the governance sub-ledger. A receipt therefore includes the ledger index of the last governance transaction, and clients must obtain the receipt of this governance transaction and all those preceding it. Clients cache governance transaction receipts and fetch missing ones from replicas.

❹ **Auditing** returns a *universal proof-of-misbehavior* (uPoM) if clients obtain receipts that are inconsistent with a linearizable execution. IA-CCF's ledger is universally-verifiable, i.e., anyone can act as an *auditor*: they replay the ledger, check consistency with receipts, and potentially generate a uPoM.

Since all consortium members and replicas may misbehave, an *enforcer*, e.g., a court, must compel members to produce a ledger copy for auditing, sanctioning non-compliance. The enforcer also punishes members based on uPoMs. It is unreasonable to assume that courts could run the service or audit long executions. Therefore, IA-CCF only requires enforcers to re-execute transactions between two consecutive checkpoints to verify a uPoM in the worst case.

After a client passes a sequence of receipts and the governance sub-ledger to the auditor, the auditor confirms the receipts' validity by calculating a Merkle tree root and verifying the replica signatures. It then asks the enforcer to obtain the ledger fragment corresponding to the receipts from the replicas. The auditor checks the validity of the checkpoint cp referenced by the oldest receipt. It then replays the ledger from cp, re-executing transaction requests while checking for consistency with receipts (including governance transaction receipts). If an inconsistency is found at index i, the auditor creates a uPoM $\langle i, \mathcal{F}, cp, R \rangle$ with a ledger fragment $\mathcal{F}$, the checkpoint cp, and the inconsistent receipt R. The uPoM is then forwarded to the enforcer, which imposes penalties on the consortium members blamed.

Threat model, and limitations. We assume a strong attacker that can compromise replicas, clients, auditors, and members to make them behave arbitrarily, but cannot break the cryptographic primitives. We trust the enforcer to assign blame to replicas and the members that operate them only when it verifies a valid uPoM or fails to obtain data for auditing. IA-CCF provides linearizability and liveness if fewer than $1/3$ of the replicas are compromised [17]. With any number of compromised replicas, clients, auditors, and members, IA-CCF never punishes members that operate only correct replicas unless they fail to provide data for auditing. In addition, IA-CCF guarantees that at least $1/3$ of the replicas are blamed, and the members that operate them punished, if clients obtain receipts that are inconsistent with a linearizable execution. The current implementation does not prevent attacks that overwhelm the ledger with transactions to slow down auditing or replaying the governance sub-ledger. It also does not blame replicas for liveness violations, e.g., not returning receipts. Possible defences include: having the enforcer timestamp the genesis transaction and bounding the rate of regular and governance transactions; and forwarding requests to the enforcer and having it monitor protocol execution to assign blame to replicas when receipts are not returned before a deadline. We leave the details of these defences for future work.

3 L-PBFT protocol and receipts

Next, we describe how L-PBFT maintains a ledger with transactions and evidence (§3.1), and how it handles view changes (§3.2). We then explain how evidence is used to create receipts (§3.3) and introduce performance optimizations (§3.4). For ease of presentation, we first assume a fixed replica set; we add dynamic membership in §5.

3.1 Protocol

To support auditing, a BFT state machine replication protocol, such as PBFT [17], must integrate with a ledger: it must ensure that replicas agree on a ledger with both transactions (requests and results) and protocol messages. It must also handle non-determinism to enable replaying the ledger. L-PBFT addresses this issue by agreeing on non-deterministic inputs [18] and using *early execution*: it requires the primary replica to propose a transaction result,

Alg. 1: Ledger Practical Byzantine Fault Tolerance

```
1  on receiveTransactionRequest (t = ⟨request,a,c,H(gt),mᵢ⟩_σ_c)
2    Pre: verify(t)
3    T ← T ∪ {t}
4  on sendPrePrepare ()
5    Pre: isPrimary () ∧ ready ∧ |T| > 0 ∧ hasEvidence (M,v,s−P)
6    B ← []; G ← {}
7    foreach t ∈ T do
8      B ← B||H(t); ⟨i,o⟩ ← execute (kv,t); G ← G||⟨t,i,o⟩
9      ⟨E_{s−P},P_{s−P},K_{s−P}⟩ ← getEvidence (M,v,s−P)
10     L ← L||P_{s−P}||K_{s−P}; M ← M||P_{s−P}||K_{s−P}
11   K[v,s] ← createNonce (); M̄ ← getRoot (M); Ḡ ← getRoot (G)
12   pp = ⟨pre-prepare,v,s,M̄,Ḡ,H(K[v,s]),E_{s−P}⟩_σ_r
13   L ← L||pp||G; M ← M||pp; M ← M ∪ {pp}; T ← {}; s ← s+1
14   sendToAllReplicas (pp||B)
15 on receivePrePrepare (pp = ⟨pre-prepare,v,s′,M̄,Ḡ,H(k),E_{s′−P}⟩_σ_r,B)
16   Pre: isBackup () ∧ verify (pp) ∧ ready ∧ s′ = s ∧ K[v,s] = nil ∧
            hasRequests(T,B) ∧ hasEvidence (M,s′−P,E_{s′−P})
17   M ← M ∪ {pp}; G ← {}
18   foreach h ∈ B do
19     t ← removeTx(h,T); ⟨i,o⟩ ← execute (kv,t); G ← G||⟨t,i,o⟩
20     ⟨E_{s−P},P_{s−P},K_{s−P}⟩ ← getEvidence (M,v,s−P,E_{s−P})
21     L ← L||P_{s−P}||K_{s−P}; M ← M||P_{s−P}||K_{s−P}
22   if getRoot (M) ≠ M̄ or getRoot (G) ≠ Ḡ then
23     undo(pp,kv,M,B,T,L); return
24   L ← L||pp||G; M ← M||pp; K[v,s] ← createNonce ()
25   p = ⟨prepare,r,H(K[v,s]),H(pp)⟩_σ_r
26   sendToAllReplicas (p); M ← M ∪ {p}; s ← s+1
27 on receivePrepare (p = ⟨prepare,r′,H(k_{r′}),H(pp)⟩_σ_{r′})
28   Pre: verify (p)
29   M ← M ∪ {p}
30 on batchPrepared (pp = ⟨pre-prepare,v,s′,M̄,Ḡ,H(k_p),E_{s′−P}⟩_σ_p)
31   Pre: prepared(pp,M) ∧ ∃⟨prepare,r′,H(K[v,s′]),H(pp)⟩_σ_{r′} ∈ M
32   c = ⟨commit,v,s′,r,K[v,s′]⟩
33   sendToAllReplicas (c); M ← M ∪ {c}
34   foreach ⟨t,i,o⟩ ∈ getTxForBatch (L,v,s′) do
35     sendReplyToClient (t,⟨reply,v,s′,r,σ_r,K[v,s′]⟩)
36     if shouldSendReceipt (r,t) then
37       S ← getMerklePath (G,i)
38       sendReceiptToClient (t,⟨replyx,v,s′,M̄,H(k_p),E_{s′−P},H(t),i,o,S⟩)
39 on receiveCommit (c = ⟨commit,v,s′,r′,k_r⟩)
40   Pre: verify (c)
41   M ← M ∪ {c}
```

which the backup replicas must agree on for the batch to commit. L-PBFT then maintains Merkle trees over all ledger entries and puts the trees' roots in protocol message, which ensures that all replicas agree on a serial history of the ledger.

Fig. 2 gives an overview of L-PBFT with early execution: first clients send transaction requests to all replicas. The primary orders the requests, groups them into *batches* and performs early execution. It then sends a pre-prepare message to the backups, which includes the request batch and the execution results. Upon receiving the pre-prepare, the backups execute the requests and confirm that the results match the primary's. If so, they send a prepare message to all other replicas. After a replica receives a pre-prepare and $N-f-1$ matching prepare messages for the same sequence number s and view v, the batch is *prepared* at the replica at v with s if all batches with lower sequence numbers have also prepared. A replica then sends a reply to the clients and commit messages to the other replicas. We say that a batch is *committed* at sequence number s if it has been prepared by $N-f$ replicas in the same view. A client has received a complete response when it has a *receipt* consisting of replies from $N-f$ replicas.
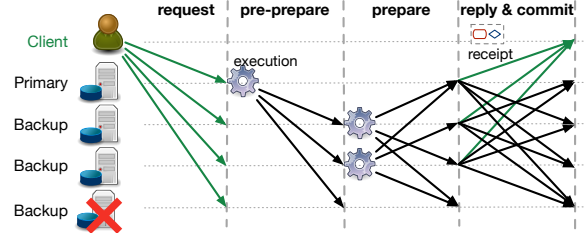
Fig. 2: L-PBFT protocol with early execution and receipts

A naive approach would require each replica to sign two protocol messages, i.e., the pre-prepare/prepare and the commit message, for each committed batch. Instead, L-PBFT uses a novel *nonce commitment* scheme, in which replicas only sign the pre-prepare/prepare messages after including a hashed nonce. Instead of signing the commit, a replica includes the unhashed nonce. This effectively halves the signatures that replicas emit to commit batches successfully.

Alg. 1 presents the pseudocode of L-PBFT. The replica state includes: the current view v and batch sequence number s; a set of transaction requests T waiting to be ordered; a message store M; a nonce store K; a boolean ready indicating if the replica can send/accept pre-prepare messages; a replica identifier r; the key-value store kv; the ledger L; and the Merkle tree M that binds the ledger entries.

In receiveTransactionRequest (line 1), a replica adds a request message to T, where a identifies the invoked stored procedure and its arguments, c is the client identifier, $H(gt)$ is the genesis transaction hash, m_i is the minimum index after which the request can be added to the ledger, and σ_c is the client signature. σ_c and $H(gt)$ ensure that requests cannot be forged or moved to a different ledger, and m_i allows clients to create an ordering dependency between the request and a previously executed transaction.

The function sendPrePrepare (line 4) uses early execution to include the execution result in the batch's Merkle tree root. The primary $p = v \bmod N$ collects a batch of transaction requests, executes them, and appends them to a new Merkle tree G. Then, the primary retrieves the commitment evidence P_{s-P} and K_{s-P} for the batch at $s-P$ from the message store M and appends it to the ledger. E_{s-P} is a bitmap that records the replicas that supplied commitment evidence.

Next, the primary creates the pre-prepare message with the hash of a fresh nonce $K[v,s]$, the root of the Merkle trees, $\bar{M}$ and $\bar{G}$, and signs it. G is a Merkle tree that contains all $\langle t,i,o \rangle$ entries in a batch. The complete pre-prepare message has two extra fields: i_g, the index of the last governance transaction, which allows clients to verify receipts with a changing set of replicas (see §5.2); and d_C, a digest of the key-value store state at the last checkpoint, which enables auditing from a checkpoint without replaying the ledger from the start (see §4).

By signing $\bar{M}$, the primary commits to the contents of the ledger, including the commitment evidence for $s-P$ that it retrieved and added to the ledger. It is important for the primary to order the evidence to ensure that replicas agree on the
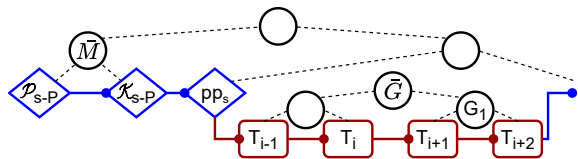
Fig. 3: Ledger with evidence and Merkle trees

ledger: if replicas added their own evidence to the ledger when they received prepare and commit messages, their ledgers could diverge. The commitment evidence $\mathcal{P}_{s-P}$ contains $N-f-1$ prepare messages for sequence number $s-P$ and view v that match the pre-prepare at sequence number $s-P$ in the ledger. $\mathcal{K}_{s-P}$ are the $N-f$ nonces with hashes in the pre-prepare/prepare messages in $\mathcal{P}_{s-P}$. This evidence is sufficient to prove to a third party that the batch at $s-P$ prepared at $N-f$ replicas and therefore committed with s. The pre-prepare message along with the leaves of G are then added to the ledger.

The primary communicates its ordering decision by sending the pre-prepare message to all replicas, together with a list $\mathcal{B}$ of the hashes of transaction requests in execution order. The requests are sent separately by the clients, and the commitment evidence for $s-P$ is not included in the message. The pre-prepare messages are $O(N)$ in size but the constant is small. Our implementation uses 8 bytes in the E_{s-P} bitmap to support up to 64 replicas, making the pre-prepare messages effectively $O(1)$.

Fig. 3 gives an example of the ledger state after this step. For each transaction in the batch, the primary adds a ledger entry in the order executed. The entry for T_i has the form $\langle t,i,o\rangle$ where o includes the reply sent to the client and the hash of the transaction's write-set; pp_s is the pre-prepare for s, and $\mathcal{P}_{s-P}$ and $\mathcal{K}_{s-P}$ are evidence that the batch at sequence number $s-P$ committed. L-PBFT pipelines the ordering of up to $P \geq 1$ concurrent batches to improve performance. Therefore, the commitment evidence lags P behind s, because it is unavailable when the primary sends the pre-prepare for s. Appx. A, Lemma 2 shows that *early execution* maintains linearizability.

When a backup replica receives the pre-prepare (line 15), it rejects the message if it already sent a prepare for the same view and sequence number ($\mathcal{K}[v,s] \neq$ nil). Otherwise, it checks if it already has the requests and commitment evidence referenced by the pre-prepare. Replicas store received requests, prepare, and commit messages in non-volatile storage ($\mathcal{M}$) until they receive (or send) a corresponding pre-prepare. To reduce network load, the primary does not resend requests or messages used as commitment evidence. If the backup is missing messages, it requests that the primary retransmit them, because a correct primary is guaranteed to have them.

The backup then executes the requests in the order prescribed by the primary, and adds the resulting transaction entries to a new Merkle tree G (line 19). Then, it adds the same $\mathcal{P}_{s-P}$ and $\mathcal{K}_{s-P}$ as the primary to the ledger. At this point, the ledger at the backup should be identical to the one at the primary just before the pre-prepare message is added. The backup checks that the roots of its Merkle trees match $\bar{M}$ and

$\bar{G}$ in the pre-prepare, respectively. If not, the message is rejected, the entries for batch s are removed from the ledger, and the transactions are rolled back. Otherwise, the backup adds the pre-prepare to the ledger, followed by the leaves of the Merkle tree G, and sends a matching prepare message with the format $\langle\text{prepare},r,H(\mathcal{K}[v,s]),H(pp)\rangle_{\sigma_r}$, where $H(\mathcal{K}[v,s])$ commits a fresh nonce, and $H(pp)$ is the pre-prepare's hash.

L-PBFT ensures deterministic transaction execution by agreeing on non-deterministic inputs [18]. Line 22 ensures that a backup's execution of batch $\mathcal{B}$ and its ledger are identical to those of the primary by comparing the Merkle roots $\bar{G}$ and $\bar{M}$. If this check fails, the backup rolls back execution and attempts to view change (§3.2). This way divergent execution due to bugs, i.e., failing to identify non-deterministic inputs, can affect liveness but not diverge the ledger.

In batchPrepared (line 30), the nonce commitment and early execution allow replicas to return replies to clients in two message round trips without signing reply or commit messages. When the batch prepares at replica r, it sends a commit message with the format $\langle\text{commit},v,s',r,\mathcal{K}[v,s']\rangle$ where $\mathcal{K}[v,s']$ is the nonce the replica committed to in the pre-prepare/prepare messages that it sent for v and s'. Since the nonce $\mathcal{K}[v,s']$ is revealed to clients and replicas only when a replica prepares the batch having a pre-prepare/prepare message and the corresponding nonce can prove to a third party that the replica prepared the batch at v and s' (see Appx. A, Lemma 3).

Finally, a replica r commits a prepared batch v,s' after it receives $N-f$ commit messages, including its own. The nonce hashes in the commit messages must match the ones in the pre-prepare/prepare messages.

We prove that L-PBFT produces a linearizable execution order in Appx. A, Thm. 1.

3.2 View changes

During the L-PBFT protocol execution, the primary may misbehave or be slow, which requires a *view change*. The change of the primary must be done in a manner that does not preclude auditing, which is a new requirement that goes beyond PBFT's view change protocol. L-PBFT view changes are auditable and must provide proof that a batch's re-execution produces the same result as the original execution.

L-PBFT addresses this as follows: it sends the evidence that batches prepared during view changes and includes the Merkle tree root $\bar{G}$ of a batch and its execution in the *pre-prepare* message, which ensures that batches are re-executed consistently. During a view change, each replica sends a view-change message with information about prepared requests. The primary for a new view v' sends a new-view message backed by $N-f$ view-change messages for v'. For each sequence number with a prepared batch in the view-change messages, the primary picks the batch that prepared with the largest view and proposes it in v'. Since all committed requests have also prepared, this ensures linearizability with batch execution ordered by the sequence

Alg. 2: View Changes in L-PBFT

```
1  on sendViewChange ()
2     Pre: primaryAppearsFaulty(v)
3     PP = getPLastPrepared(msgs(L) ∪ M)
4     v = v+1; ready ← false; vc = ⟨view-change,v,r,PP⟩_σ_r
5     sendToAllReplicas (vc) ;  M ← M ∪ {vc}
6  on receiveViewChange (vc = ⟨view-change,v',r',PP⟩_σ_r )
7     Pre: v' >= v ∧ verify(vc) ∧ hasPrepares (msgs(L) ∪ M,getLast(PP))
8     M ← M ∪ {vc}
9     if |getViewChanges(M,v')| > f ∧ v' > v then
10       │  v = v'−1; setPrimaryApearsFaulty()
11       │  sendViewChange ()
12 on sendNewView (v)
13    Pre: isPrimary (v) ∧ ¬ready ∧ |getViewChanges(M,v)| > N−f
14    ⟨M̄,E_vc,h_vc,PP_ov⟩ = processViewChanges(getViewChanges(M,v))
15    nv = ⟨new-view,v,M̄,E_vc,h_vc⟩_σ_r ; L ← L || nv; M ← M || nv
16    sendToAllReplicas (nv)
17    resendPreparesInNewView(PP_ov); ready ← true
18 on receiveNewView (nv = ⟨new-view,v,M̄,E_vc,h_vc⟩_σ_r ,PP_mv)
19    Pre: isPrimary (r',v)
             ∧ hasRequests (T,PP_mv) ∧ hasEvidence (M,PP_mv)
             ∧ r' ≠ r ∧ ¬ready ∧ |getViewChanges(M,E_vc,h_vc)| > N−f
20    ⟨M̄',PP'_ov⟩ = processViewChanges(getViewChanges(M,E_vc,h_vc))
21    if M̄' = M̄ then
22       │  L ← L || nv; M ← M || nv
23       │  if ready ← processPreparesInNewView(PP_mv,PP'_ov) then return
24    undo(nv,s,M,L)
```

Alg. 3: Verifying Receipts

```
1  on verifyReceipt (⟨t,i,o⟩,⟨v,s,M̄,H(k_p),E_{s−P},i_g,d_C⟩,σ_p,E_s,Σ_s,K,S⟩)
2     Ḡ' ← pathHash (⟨t,i,o⟩)
3     foreach G_i ∈ S do
4       │  Ḡ' ← pathHash (Ḡ', G_i)
5     pp = ⟨pre-prepare,v,s,M̄,Ḡ',H(k_p),E_{s−P},i_g,d_C⟩
6     if not checkSignature (σ_p, pp) then   return false
7     foreach r ∈ E_s do
8       │  if r = p ∧ H(K[p]) ≠ H(k_p) then   return false
9       │  if r ≠ p ∧ not checkSignature
             (Σ_s[r], ⟨prepare,r,H(K[r]),H(pp_{σ_p})⟩) then   return false
10    return true
```

by adding an entry with the $N-f$ view-change messages that it accepted to the ledger in order of increasing replica identifier; h_{vc} is the hash of that entry and E_{vc} is a bitmap with the replicas that sent the messages. It returns the root of the Merkle tree $\bar{M}$, E_{vc}, h_{vc}, and PP_{ov} (line 14). The primary appends the new-view to the ledger, sends it to all replicas, resends the prepared batches in pre-prepare messages in the new view, and adds them to the ledger.

When backups receive the new-view (line 18), they obtain missing view-change messages, requests and evidence that it references, and call processViewChanges. If it returns a Merkle tree root equal to the one in new-view, they accept the message, add it to the ledger, and process the pre-prepare messages PP_{nv}. If these match the batches and evidence in PP'_{ov} for the same sequence numbers, they are added to the ledger; otherwise, all changes are undone.

3.3 Receipts

To allow third parties to audit the ledger against the transaction results returned to clients, L-PBFT returns *receipts*, which are statements signed by $N-f$ replicas that a transaction request t executed at index i and produced a result o. L-PBFT exploits the per batch Merkle tree G together with the nonce commitment scheme (§3.1) to avoid having replicas sign the reply for each request.

Creating receipts. When a transaction batch described by pre-prepare pp prepares at replica r, view v and sequence number s' (Alg. 1, line 30), it sends $\langle \text{reply},v,s',r,\sigma_r,K[v,s']\rangle$ to every client with a transaction in the batch. (If the client has multiple transactions in the batch, only one reply is sent.) By revealing the nonces, the replicas provide the client with proof that they claimed to have prepared the batch without a signed reply.

Only a designated replica, chosen based on t, sends the result and the rest of the receipt to the client (line 36). The replica computes a list of sibling hashes S along the path from the leaf to the root of the per-batch Merkle tree G. For the example of T_i in Fig. 3, S consists of the digest of T_{i-1} and G_1, which is sufficient to recompute $\bar{G}$ given T_i. It then sends the client $\langle \text{replyx},v,s',\bar{M},H(k_p),E_{s'-P},i_g,d_C,H(t),i,o,S\rangle$, where i_g and d_C are used for auditing.

Verifying receipts. The client waits for $N-f$ replicas to send reply messages with the same v and s, and for a replyx message with the same v and s. It then recreates the pre-prepare

numbers at which batches committed.

Alg. 2 formalizes the pseudocode for view changes. If the primary for view v appears faulty or slow, a replica sends a view-change message, $\langle \text{view-change},v+1,r,PP\rangle_{\sigma_r}$, to all other replicas (line 1), where PP contains the last P pre-prepare messages that prepared locally (line 3). Only the last message in PP is required to provide linearizability, because it includes the Merkle tree roots $\bar{M}$ and $\bar{G}$ that determine the ledger contents up to that point. The other pre-prepare messages are used during auditing to verify that replicas reported the batches they prepared in view-change (§4).

When replicas receive a view-change message (line 6), before processing it, they fetch missing prepare messages from the sender to prove that the last pre-prepare in PP has prepared. When replicas increment v, they set ready to false (lines 4, 11), which ensures that they do not send or accept pre-prepare messages until they have completed the new-view.

After accepting $N-f$ view-change messages for the new view (line 12), the new primary calls processViewChanges, which picks the view-change message vc_{lp} with the last prepared pre-prepare message pp_{lp} from those with the largest view number. It then updates the ledger to match the Merkle roots in pp_{lp} by fetching missing ledger entries from replicas that sent matching prepare messages. Since at least $f+1$ of those are correct, this is always possible. The primary checks that all messages in PP of vc_{lp} appear at the right ledger positions; if not, it discards vc_{lp} and re-tries (omitted from Alg. 2).

Next the primary resets the ledger to $s_{lp}-P$, because the batches up to this point are guaranteed to have committed. It saves all the request batches and commitment evidence for sequence numbers between $s_{lp}-P$ and s_{lp} and returns it in PP_{ov}. This is needed to resend pre-prepare messages for the prepared batches in the new view. The function ends

and prepare messages (Alg. 3, line 6), with the information in replyx and the hashes of the nonces, and verifies the signatures. (We describe how to determine N and verify signatures under dynamic membership in §5.2.) This step is shared across all transaction requests that the client may have sent in the batch.

IA-CCF uses the Merkle tree G to bind signatures in pre-prepare and prepare messages to transactions in the batch, enabling replicas to produce a single signature per batch. In the example in Fig. 3, the client checks if $\bar{G} = H(H(H(T_{i-1})\|H(\langle t, i, o\rangle))\|G_1))$ (lines 2–4). If the hashes match, the client has a valid receipt, i.e., a statement signed by $N-f$ replicas that a request t executed at index i and produced a result o; otherwise (or if the client does not receive replies before a timeout), it retransmits the request and selects a different replica to send back replyx. (The application is responsible for ensuring exactly-once semantics if needed.)

Clients store the receipt for $\langle t, i, o \rangle$ as $\langle v, s, \bar{M}, H(k_p), E_{s-P},$ $i_g, d_C, \sigma_p, E_s, \Sigma_s, \mathcal{K}_s, \mathcal{S} \rangle$ where Σ_s is a list of the signatures in prepare messages, $\mathcal{K}_s$ is a list of nonces, and E_s is a bitmap indicating the replicas with entries in Σ_s, and $\mathcal{K}_s$, sorted in increasing order of replica identifier. All receipt components, including common hashes in $\mathcal{S}$, are shared across requests in the same batch.

Clients must store the receipts together with the transaction request and the corresponding result to resolve future disputes. This is not a burden because receipts are concise: all components have constant size, except $|\mathcal{S}|$, whose number of entries is logarithmic in the number of requests in a batch; Σ_s and $\mathcal{K}_s$ have up to $N-f$ entries. In addition, most intermediate hashes in $\mathcal{S}$ can be shared across collections of receipts. We explored using signature aggregation [13] to reduce the size of Σ_s, but, for realistic consortia sizes, verifying the signatures becomes more expensive than our current implementation.

3.4 Performance optimizations

L-PBFT includes several optimizations to improve transaction and auditing throughput.

Checkpoints in L-PBFT allow new replicas to start processing requests without having to replay the ledger from the start (§5.1); slow replicas to be brought up-to-date using a recent checkpoint; and auditing to start from a checkpoint instead of the beginning of the ledger(§4.1).

Checkpoints include the key-value store and the Merkle tree M's newest leaf, root, and the connecting branches. Replicas create a checkpoint cp_s when they execute a batch with sequence number s such that $s \bmod C = 0$. The primary adds a batch to the ledger at sequence number $s+C$ with a special *checkpoint transaction*, which records the checkpoint digest. C is chosen to give replicas enough time to complete a checkpoint without delaying L-PBFT execution. Backups only accept the pre-prepare for $s+C$ if they compute the same checkpoint digest for sequence number s.

When a replica fetches checkpoint cp_s, it also retrieves the ledger up to s. It does not need to replay the ledger or check all signatures (with the exception of governance transactions; §5.2). Instead, it checks the signatures in checkpoint receipts and that the ledger contents between consecutive checkpoints are consistent with the Merkle tree roots in the corresponding receipts. This is done from the start of the ledger until $s+C$.

Cryptography. L-PBFT reduces the impact of cryptographic operations. Signature verification is parallelized for messages received from replicas and clients [12, 20] to improve throughput and scalability. All messages are sent over encrypted and authenticated connections, even signed messages. This mitigates denial-of-service attacks that consume replica resources verifying signatures [20].

To further improve performance, backups overlap the execution of request batches with the validation of pre-prepare signatures. They only send the prepare after both completed. Since pre-prepare messages are received over authenticated connections, this always succeeds for correct primaries.

4 Auditing and enforcement

In this section, we describe how auditing produces *universal proofs-of-misbehavior* (uPoMs) when linearizability is violated (§4.1), and the role of the enforcer in obtaining ledgers for auditing and punishing the members responsible for misbehaving replicas (§4.2). We first focus on the simpler case of auditing without governance transactions; §5 describes governance transactions and their impact on auditing.

4.1 Auditing

An audit is triggered when someone, usually a client, obtains a sequence of transaction receipts that violate linearizability, i.e., when no linearizable execution of the stored procedures that define the transactions can produce the sequence of receipts. The mechanism to detect linearizability violations is application dependent. It involves clients, which interact through a sequence of transactions, exchanging receipts and using the application semantics to reason about the correctness of the receipt sequence. We describe a banking-inspired example in the introduction.

The goal of auditing is to detect dishonest behavior regardless of the number of misbehaving replicas, i.e., it must find proof of misbehavior even if all replicas collude and rewrite the ledger. IA-CCF therefore tightly integrates the ledger with receipts—even if the ledger is rewritten, the misbehaving replicas are unable to alter the receipts.

An audit can be performed by anyone, and begins when an *auditor* receives a collection of receipts. Next, the auditor requests a *checkpoint* and a *ledger fragment* that contains the section of the ledger spanning the receipts. Any honest replica that signed the receipts is guaranteed to have the checkpoint and ledger fragment. When the auditor receives the requested data, it verifies the ledger structure by checking the protocol messages and their order, and validating any signatures in the ledger—but it does not re-execute transactions. Then, the auditor checks that the transactions referenced by the receipts

are present at the right positions in the ledger.

If the above steps have not discovered misbehavior, there remains the possibility that at least $N-f$ of the replicas colluded and agreed on an incorrect execution result. Therefore, the auditor loads the checkpoint and replays the transactions from the ledger fragment to check if execution results are correct. Throughout this process, if dishonest behavior is uncovered, the auditor can produce a universally-verifiable proof that at least $f+1$ replicas misbehaved.

More formally, Alg. 4 presents the pseudocode for the auditing process. First, the auditor receives an ordered set of receipts $\mathcal{R} = \{\langle\langle t_0, i_0, o_0\rangle, x_0\rangle, ..., \langle\langle t_k, i_k, o_k\rangle, x_k\rangle\}$ where $k \geq 1$ and $\forall l \in [0,k): s_l \leq s_{l+1}$. Here, s_i is the sequence number that is specified in x_i. The auditor invokes auditReceipts (line 2) to check if the receipts are valid and the minimum index requirements have been satisfied. If there is a receipt that violates the requirement in the request, all replicas that have signed the receipt can be blamed.

After that, the auditor must obtain a ledger fragment and checkpoint that are *complete* in relation to $\mathcal{R}$ (line 3). We formally define completeness in Appx. B, but intuitively the ledger fragment must be (i) *well-formed*; (ii) include all batches and evidence between sequence numbers s_{C_0} and s_k where s_{C_0} is the sequence number of the checkpoint transaction that is linked in the first receipt; and (iii) include view-change messages for all views in $\mathcal{R}$. The transaction and checkpoint at s_{C_0} must match the checkpoint linked in the first receipt. A ledger fragment is *valid* if it can be produced by a sequence of correct primaries in a sequence of views where there are at most f Byzantine failures. It is well-formed if it is valid, or if it would be valid if not for the incorrect execution of some transactions and/or checkpoints. A correct replica always maintains a well-formed ledger.

In getCheckpointAndLedger (line 3), the auditor, with the help of an *enforcer*, obtains ledger fragments and checkpoints from replicas that signed the latest receipt with the highest view number in $\mathcal{R}$ (line 10). The auditor checks if responses are complete in relation to the receipts. If a ledger fragment is not well-formed or misses the required view-change messages, the auditor can blame the responding replica. Below, we assume that the responses contain no invalid signatures, we show in Appx. B how the auditor handles that case.

If the batch at s_{C_0} is not a checkpoint or the checkpoint digest does not match the first receipt, the auditor can assign blame to the intersection of replicas that have signed the batch at $s_{C_0} + C$ and the first receipt, as the checkpoint reference in a receipt must always link to the last committed checkpoint. If the fragment is not long enough to include the sequence number in one of the receipts, there must be misbehavior during a view change. The auditor can then blame at least $f+1$ misbehaving replicas: the intersection of the replicas that participated in a view change and that also signed the receipt. A correctness proof and the details of obtaining a complete ledger fragment and checkpoint are

---

Alg. 4: Ledger Auditing (simplified)

```
1  on audit (R = {⟨⟨t₀,i₀,o₀⟩,x₀⟩,...,⟨⟨tₖ,iₖ,oₖ⟩,xₖ⟩})
2     auditReceipts (R)
3     C₀,s_C₀,L←getCheckpointAndLedger (x₀,xₖ)
4     verifyReceiptsInLedger (R,L)
5     replayLedger (C₀,s_C₀,L)
6  on auditReceipts (R = {⟨⟨t₀,i₀,o₀⟩,x₀⟩,...,⟨⟨tₖ,iₖ,oₖ⟩,xₖ⟩})
7     foreach ⟨⟨tᵢ,iᵢ,oᵢ⟩,xᵢ⟩ ∈ R do
8        if not verifyReceipt (⟨tᵢ,iᵢ,oᵢ⟩,xᵢ) then return invalidReceipt
9  on getCheckpointAndLedger (x₀,xₖ)
10    for C₀,s_C₀,L,r←enforcerGetLedgerPackage (x₀,xₖ) do
11       uPoM←nil
12       foreach s∈s_C₀,...,seqno(xₖ+P) do
13          if not isBatchWellformed (L,s) then
14             F←createLedgerFragment (nil,s,L)
15             uPoM←⟨nil,F,r⟩; send(uPoM); return
16       if uPoM=nil then return C₀,s_C₀,L
17 on verifyReceiptsInLedger (R,L)
18    foreach ⟨⟨tᵢ,iᵢ,oᵢ⟩,xᵢ=⟨v,s,H(kₚ),...K,S⟩⟩∈R do
19       if not isReceiptInBatch(xᵢ,L) then
20          F←createLedgerFragment (nil,s,L)
21          uPoM←⟨F,⟨⟨tᵢ,iᵢ,oᵢ⟩,xᵢ⟩⟩; send(uPoM); return
22 on replayLedger (C₀,s_C₀,L)
23    s_cp←s_C₀; cp←C₀; kv←loadCheckpoint(s_C₀,C₀)
24    foreach s∈s_C₀,...,seqno(xₖ) do
25       foreach ⟨tᵢ,iᵢ,oᵢ⟩∈s do
26          L,kv←replayRequest(L,kv,tᵢ)
27          if not verifyReplay (L,kv,⟨tᵢ,iᵢ,oᵢ⟩) then
28             F←createLedgerFragment (s_cp,s,L)
29             uPoM←⟨iᵢ,F,cp⟩; send(uPoM); return
30       if s mod C=0 then
31          s_cp←s; cp←createCheckpoint(kv)
```

described in Appx. B, Lemmas 4 and 6.

After obtaining a well-formed ledger, in verifyReceiptsInLedger (line 4), the auditor compares the receipts with the ledger. If a receipt $\langle\langle t_k, i_k, o_k\rangle, x_k\rangle$ does not match the batch at s_k in the ledger fragment, we show in Lemma 5 that the auditor can assign blame to $f+1$ misbehaving replicas. In summary, there are three cases: (i) the pre-prepare with sequence number s_k in L has a view number $v_l = v_k$; (ii) $v_l > v_k$; or (iii) $v_l < v_k$. In case (i), the ledger fragment contains evidence that the batch with sequence number s_k has prepared at $N-f$ replicas. Since at least $f+1$ of the replicas that have prepared the batch also signed the receipt, they can be blamed. In case (ii), since $v_l > v_k$, there must be at least $N-f$ view-change messages from different replicas that transition to a view greater than v_k in the ledger fragment but claim not to have prepared the batch in the receipt in view v_k. Since there are at least $f+1$ of those replicas that also signed the receipt, they can be blamed. In case (iii), since $v_k > v_l$ and the ledger fragment is complete in relation to the receipt, there must be at least $N-f$ view-change messages from different replicas that transition to a view greater than v_l in the ledger fragment. Similarly, the intersection of those replicas and the ones that signed the receipt can be blamed.

Since $N-f$ or more replicas may have misbehaved, it is necessary to replay transaction execution to check if the results are correct. The auditor does not need to understand the semantics of the service; it can retrieve the code of the stored procedures from C_0. The auditor sets the service state to the checkpoint value and replays transactions. If replaying a transaction fails to match the result in the ledger, the auditor can

assign blame to any replica that signed the batch that contains the transaction. This is shown in replayLedger (line 5).

4.2 Enforcement

Since IA-CCF provides individual accountability even if all replicas and members misbehave, there must be an *enforcer* outside of the system to obtain checkpoints and ledger fragments for auditing, and to punish members responsible for misbehaving replicas. For example, consortium members may sign a binding contract to establish penalties if a uPoM proves that one of their replicas misbehaved, or if they fail to produce checkpoints and ledgers for auditing by an agreed deadline. These penalties may be imposed by the enforcer via arbitration [8] or a court of law [9].

The enforcer receives a set of receipts $\mathcal{R}$ from the auditor (Alg. 4, line 10). It then verifies that the receipts are valid, and requests all of the replicas that signed the latest receipt with the highest view for a ledger fragment that is complete in relation to $\mathcal{R}$.

Correct replicas will respond to the enforcer quickly. If the enforcer does not receive a response from a replica within a reasonable duration, e.g., within minutes, it contacts the controlling consortium member to obtain the checkpoint and ledger. If the member fails to provide this information by an agreed deadline, e.g., within days, it is punished according to the contract. This is important to ensure that misbehaving members cannot escape punishment by failing to produce information for auditing. However, it introduces a weak synchrony assumption that may lead to the punishment of honest but slow members. We expect that the deadline will be chosen conservatively to make this unlikely in practice. After the deadline elapses, the enforcer either returns to the auditor $f+1$ responses, or it penalizes $f+1$ unresponsive replicas.

The enforcer also punishes members if a uPoM proves that one of their replicas misbehaved. When it receives a uPoM, it checks its validity by carrying out an audit, as described in §4.1, but the ledger fragment size and the number of transactions to replay is bounded by the transactions between two consecutive checkpoints. Furthermore, if there are fewer than $N-f$ misbehaving replicas, the uPoM does not require the enforcer to replay transactions. If the uPoM is incorrect, the enforcer punishes the auditor; otherwise, it punishes the members responsible for at least $f+1$ misbehaving replicas.

In practice, we expect the load placed on the enforcer to be small, because auditing is rare—IA-CCF provides linearizability with up to f misbehaving replicas and the enforcer penalizes entities that request information for auditing and fail to produce a valid, minimal uPoM.

5 Reconfiguration and auditing

In this section, we describe how IA-CCF can change the consortium membership and the active replica set (§5.1). We explain how this impacts receipt validation (§5.2) and auditing (§5.3).

5.1 Reconfiguration

An IA-CCF deployment must handle changes to the active member and replica set while supporting auditing, regardless of how many replicas misbehave. For this, IA-CCF maintains governance data in the form of a *configuration*, which includes the public signing keys for members and replicas and an endorsement of each replica's signing key signed by the member responsible.

Changing the configuration enables members to change the active replica set. This is initiated by a *referendum*: members propose an updated configuration followed by the other members voting on the proposal. The number of votes required to pass the proposal is part of the service's state.

When voting on proposals, members must ensure the integrity of the service, e.g., disallowing an individual member from controlling too many replicas. Members are also limited to adding or removing at most f replicas, which ensures that the configuration change does not effect the service's liveness.

A referendum is carried out through governance transactions: a member proposes a new configuration by sending a *propose* transaction request. This is followed by members sending *vote* requests. Upon executing the final *vote* transaction required for a referendum to pass at sequence number s, the primary ends the current batch, and initiates the reconfiguration process.

A *reconfiguration* first adds evidence for the referendum to the ledger. This is done as part of the old configuration by the primary sending P pre-prepare messages without batched requests, called the *end-of-configuration* batches. The pre-prepare message for the end-of-configuration batch at sequence number $s+P$ contains evidence that the batch at s committed (§3). In addition, these pre-prepare messages include an extra field: the *committed* Merkle root, which is the root of the Merkle tree at s. This evidence is required for auditing: it commits the replicas that signed the Pth end-of-configuration batch to triggering the reconfiguration. Similarly, the signatures of the replicas that prepared the Pth end-of-configuration batch must be included in the ledger in the same configuration. Following the first P end-of-configuration batches, the primary pre-prepares another set of P end-of-configuration batches. The configuration change takes effect at $s+2P$.

The replicas in the new configuration create a checkpoint of the key-value store at sequence number $s+2P$. The primary creates a pre-prepare for the checkpoint at $s+2P+1$, followed by P *start-of-configuration* pre-prepare messages with empty request batches. This ensures that a correct replica commits the checkpoint transaction before other transactions are executed in the new configuration. If any of the end/start-of-configuration batches correspond to a checkpoint sequence number, the checkpoint is skipped. Therefore, the checkpoint digests d_c in the pre-prepare messages always refer to checkpoints in the same configuration.

A newly added replica first obtains the ledger and a recent

checkpoint, and replays the ledger from that checkpoint (§3.4). Replicas that are no longer part of the new configuration retire after sending the pre-prepare for $s+2P$. Removed members and replicas should delete their private signing keys to provide forward security. This prevents them from being blamed for future compromises, while still allowing authentication of transactions in the ledger using their public keys.

5.2 Governance sub-ledger and receipts

When a client verifies a receipt, it must know which replicas were active when the receipt was created. IA-CCF addresses this with the help of the governance sub-ledger.

Governance transactions are recorded in the ledger and used by auditors to determine the active configuration. Clients, however, do not have a copy of the ledger, but need to verify receipt signatures. To do this, they store receipts for all governance transactions and, for each reconfiguration, they also store the receipts for the Pth end-of-configuration batch. We refer to this as the receipts of the *governance sub-ledger*. A client checks that a transaction receipt for index i is valid by considering the governance sub-ledger from the genesis transaction gt up to i. The client verifies the governance receipts, and if successful, the replica signing keys at index i are used to validate the receipt (§3).

This raises the challenge of how a client determines that it has *all* required governance receipts. IA-CCF includes the ledger index of the last governance transaction in each pre-prepare message and receipt (i_g). A client can request missing receipts from replicas by traversing the sequence of governance receipts. It verifies received receipts incrementally and caches them locally.

With reconfiguration, the definition of a valid receipt is extended: a valid receipt R must include valid governance receipts from gt up to the configuration that produced R.

5.3 Auditing

Reconfiguration introduces several new tasks for the auditor: it must consider the governance sub-ledger with receipts; validate that reconfigurations were executed correctly; and ensure that that only one configuration was active for any given index or sequence number. Next, we provide a summary of the required changes to the auditing process; a detailed correctness proof is included in Appx. B.2.

A client initiates an audit by sending inconsistent receipts and the supporting governance receipts to an auditor. The auditor replays these governance transactions to determine the signing keys required to verify each client receipt. After verifying the receipts, the auditor requests a ledger fragment and checkpoint from the enforcer.

The auditor may uncover that multiple configurations were active for a given index or sequence number, this can happen when misbehaving replicas fork or rewrite the ledger. We call this a *fork in governance*. If the auditor finds a fork, there are two P^{th} end-of-configuration batch receipts with the same preceding configuration that are not *equivalent*: they are at different indices or sequence numbers, or their pre-prepare messages do not contain the same committed Merkle root, i.e., they are not preceded by the same governance transactions. In this case, the auditor assigns blame to the replicas that signed both receipts, as a correct replica that prepares a Pth end-of-configuration batch commits the final *vote* transaction that triggers reconfiguration.

If the enforcer cannot obtain the required information for a valid receipt R from the sequence of provided receipts, there must be misbehaving replicas. In addition to the misbehavior described in §4.1, the misbehaving replicas may have created a fork in governance or incorrectly prepared the Pth end-of-configuration batch that succeeds the configuration that produced the receipt R (see Lemmas 8 and 11).

Another possibility is that the configuration that produced a receipt R for a sequence number s may not match the configuration that prepared the batch at s in a well-formed ledger fragment. In this case, blame is again assigned to the replicas that signed R and prepared the Pth end-of-configuration batch that succeeds the configuration that produced R (see Lemma 9).

After assigning blame, the auditor sends a uPoM to the enforcer with the supporting governance receipts.

6 Evaluation

We evaluate IA-CCF to understand the cost of providing receipts (§6.1), its scalability (§6.2), the overheads of receipt validation (§6.3), and auditing (§6.5). We finish with a performance breakdown of IA-CCF's design features (§6.8).

Testbeds. Our experimental setup consists of three environments: (a) a dedicated cluster with 16 machines, each with an 8-core 3.7-Ghz Intel E-2288G CPU with 16 GB of RAM and a 40 Gbps network with full bi-section bandwidth; (b) a LAN environment in the Azure cloud, with Fsv2-series VMs with 16-core 2.7-GHz Intel Xeon 8168 CPUs and 7 Gbps network links; and (c) a WAN environment with the same VMs across 3 Azure regions (US East, US West 2, US South Central). All machines run Ubuntu Linux 18.04.4 LTS.

Implementation. Our IA-CCF prototype is based on CCF v0.13.2 [45] and has approx. 40,000 lines of C++ code. It uses the formally-verified Merkle trees and SHA functions of EverCrypt [51], the MbedTLS library [41] for client connections, and secp256k1 [61] for all secure signatures. Replicas create secure communication channels using a Diffie–Hellman key exchange.

Pipelining batch execution (P in Alg. 1) improves IA-CCF's throughput. We use $P=2$ for the LAN and $P=6$ for the WAN, with maximum batch sizes of 300 and 800 requests, respectively. Checkpoints are created every 10K or 4K sequence numbers in the LAN and WAN environments, respectively.

Benchmarks. We use the *SmallBank* benchmark [2], which models a bank with 500K customer accounts. Clients randomly execute 5 transaction types: deposit, transfer, and withdraw funds; check account balances; and amalgamate ac-

Tab. 1: Size of ledger entries (SmallBank)

Ledger entry type	Size (bytes)	
	f = 1	f = 3
Transaction (SmallBank)	216–358	
Pre-prepare	277	
Prepare Evidence	298	894
Nonces	32	64
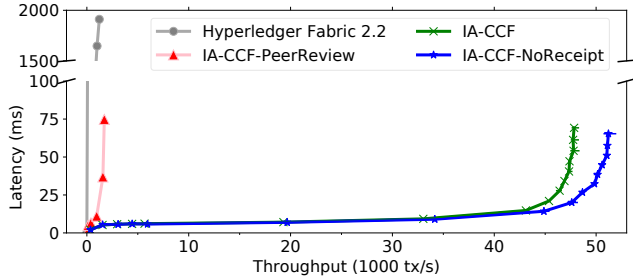
Fig. 4: Transaction throughput/latency ($f=1$, dedicated cluster)

Tab. 2: Request latency under low load (WAN)

	average latency	99th percentile latency	network round trips
IA-CCF	183 ms	194 ms	2
HotStuff	340 ms	393 ms	4.5

Fig. 5: Transaction throughput vs. replica count (WAN)

counts. The size of the ledger entries is shown in Tab. 1 where only the Prepare Evidence and Nonces entries depend on f.

Since IA-CCF's design targets accountability with more than f failures, we omit results from experiments with fewer failures. In such cases, IA-CCF's performance matches that of prior work, because it uses well-established BFT techniques, such as view changes, sending messages via authenticated channels and client-signed requests [12, 20]. Instead, we consider the performance of receipt validation (§6.3) and auditing (§6.5), which are new contributions of IA-CCF.

Transaction throughput is measured at the primary replica and latency at the clients. All experiments are compute-bound. Results are averaged over 5 runs, with min/max error bars.

Baselines. We compare against four baselines: IA-CCF-PeerReview, which uses PeerReview for accountability [27], i.e., replicas sign all messages and send signed acknowledgements for all messages; IA-CCF-NoReceipt, an IA-CCF variant that produces a ledger but no receipts; HotStuff [62], a state-of-the-art BFT protocol, which is at the core of the Diem permissioned ledger system [3]; and Hyperledger Fabric (v. 2.2) [4], a popular open-source permissioned ledger system. We compare against Fabric's latest major release that does not include a BFT consensus protocol [33] and only tolerates crash failures using Raft [49].

6.1 Transaction throughput and latency

We explore the throughput and latency of transaction execution with 4 replicas ($f=1$) in the dedicated cluster, comparing IA-CCF, IA-CCF-NoReceipt, IA-CCF-PeerReview, and Fabric.

Fig. 4 shows a throughput/latency plot as transaction load increases. IA-CCF achieves 47,841 tx/s while maintaining latencies below 70 ms. As the load increases, queueing delays increase latency. IA-CCF-NoReceipt's throughput is 51,209 tx/s, which is only 3% higher than IA-CCF, demonstrating the low cost of receipts.

IA-CCF-PeerReview exhibits an order of magnitude lower
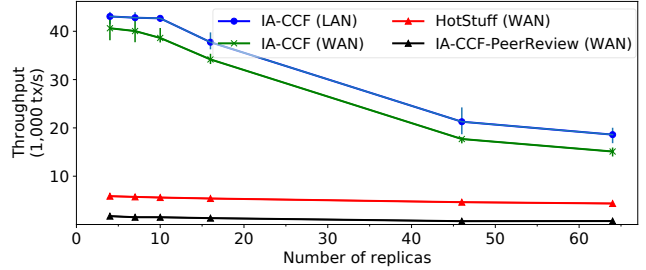
throughput because all messages must be signed, e.g., a replica must sign a reply message for each transaction in a batch. This causes IA-CCF-PeerReview to perform two orders of magnitude more asymmetric cryptographic operations than IA-CCF.

Fabric's throughput is only 1,222 tx/s, with a latency of 1.9 s. This is substantially worse than IA-CCF, despite not using a BFT protocol. Our analysis reveals two reasons: Fabric's *execute-order-validate* model requires that replicas issue a signature for each executed transaction, while IA-CCF replicas only require one signature per batch; and Fabric suffers from documented inefficiencies related to its key-value store implementation [48].

6.2 Scalability

Next we consider the effect on transaction throughput when increasing the number of IA-CCF replicas in the Azure WAN environment, spanning multiple regions to reduce correlated failures [10]. We compare against IA-CCF deployed in the Azure LAN environment, IA-CCF-PeerReview, and HotStuff, a BFT consensus protocol without a ledger or key-value store.

Fig. 5 shows that, as expected, IA-CCF's throughput decreases with more replicas because more signatures are verified by each replica. Since each replica has a fixed number of threads for checking signed pre-prepare/prepare messages in parallel, throughput decreases when the replica count exceeds the number of hardware threads, which is only 16 in this deployment. IA-CCF is only marginally affected by the higher WAN latencies due to its use of pipelining, as shown by the comparison to the LAN deployment.

HotStuff [63] achieves a throughput of 5,862 tx/s in the WAN environment, which is worse than its reported LAN throughput [66]. While it degrades slowly with more replicas, even with 64 replicas its throughput remains 71% lower than that of IA-CCF. The throughput of IA-CCF-PeerReview is even lower since it performs more cryptographic operations.

We also measure the request latency of HotStuff and IA-CCF under low load. As reported in Tab. 2, HotStuff's request latency is approximately twice that of IA-CCF's. For

(a) 100,000 accounts **(b) 500,000 accounts** **(c) 1,000,000 accounts**
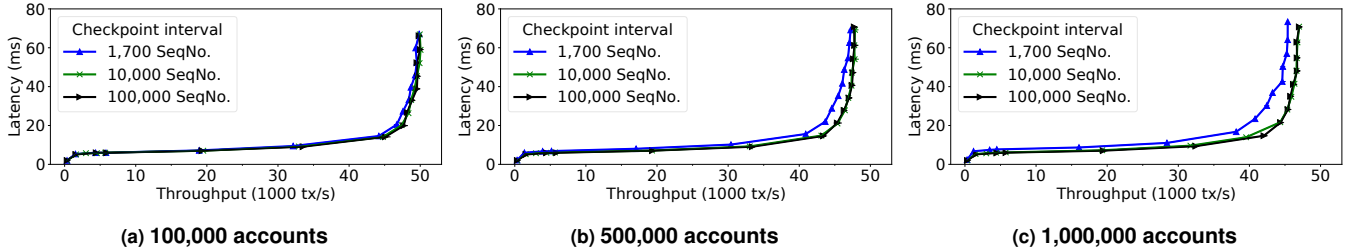
Fig. 6: Transaction throughput/latency when varying the number of accounts and checkpoint interval (f=1, dedicated cluster)

both systems, request latency is dominated by the number of network round trips and clients receive transaction results with receipts in only 2 round trips in IA-CCF.

6.3 Receipt validation

We measure the time required to verify receipts, which depends on (i) the length of the path in the Merkle tree G and (ii) the number of signatures to be checked. Since the number of leaves in G is bounded by the batch size, the path length remains small: verification takes 2.1 µs and 2.3 µs for batches of 300 and 800 requests, respectively. The overall cost is dominated by the signature verification, which takes 18 ms and 52 ms for f=1 and f=3, respectively.

6.4 Governance sub-ledger

Next, we consider the size of the governance sub-ledger, which is stored by clients. The sub-ledger is a collection of receipts for every transaction that has updated the governance of an IA-CCF deployment. A receipt's size is 623 bytes or 1,565 bytes for f=1 or f=3, respectively. In addition, the client must store the governance request and the corresponding response, which have variable size. We expect governance operations to be rare. Therefore, storing and verifying governance sub-ledger receipts has low overhead.

6.5 Ledger auditing

Next, we want to understand auditing performance. For the SmallBank workload, we compare execution time to auditing time. When measuring throughput at f=1, auditing is 23% faster than execution, because there is no network overhead, message signing, or ledger writes. In each batch, IA-CCF only verifies $2f$+1 rather than up to $3f$+1 signatures. For f=4, the performance gap increases to 67%, as more replicas add communication and cryptographic load during execution. We observe that the bottleneck for auditing is verification of client request signatures, which can be trivially parallelized.

6.6 Key-value store

We explore the performance impact of varying the number of entries in the key-value store by varying the number of Small-Bank accounts. Fig. 7 shows a throughput vs. latency plot. As expected, throughput decreases when the number of entries in the key-value store increases. CCF's implementation [54] of the key-value store uses a CHAMP map [58], whose access time grows logarithmically with the number of items.
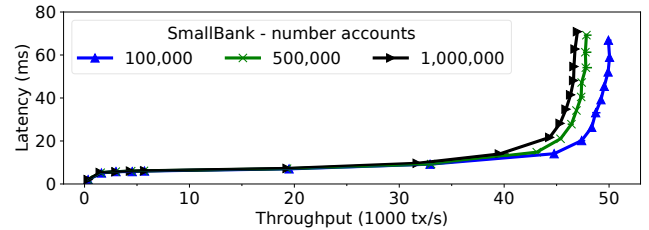
Fig. 7: Transaction throughput/latency with different account numbers (f=1, dedicated cluster)

6.7 Checkpointing

We also explore the effect of checkpointing on performance. We vary the size of the key-value store and the checkpoint interval for the SmallBank workload. Fig. 6 shows the results as throughput vs. latency plots. As expected, the checkpoint overhead increases with the size of the key-value store and the checkpoint frequency, but the overhead is low for checkpoint intervals between 10 and 100K (approximately 1 to 10 minutes). The checkpoint interval impacts the overhead to check uPoMs at the enforcer. We expect checkpointing every 10 minutes to be acceptable in practice; it requires the enforcer to replay at most 10 minutes of transactions.

6.8 Overhead breakdown

To provide a permissioned ledger with individual account-ability, IA-CCF implements functionality that goes beyond traditional BFT consensus protocols, e.g., generating receipts. We now explore the impact of implementing this functionality on IA-CCF's throughput in the dedicated cluster.

We compare several variants of IA-CCF, each limiting functionality further: (a) IA-CCF; (b) IA-CCF-NoReceipt, i.e., without creating receipts; (c) without creating checkpoints; (d) with a small key-value store, i.e., the key-value store fits in the CPU cache; (e) without signed client requests; (f) using only MACs for message authentication between replicas; (g) without a ledger; and (h) with empty requests, i.e., without the overhead of executing transactions against the key-value store.

Tab. 3 shows that (a)–(d) have comparable throughput, but not verifying client signatures (e) doubles throughput. Only using MACs instead of signatures (f) or removing the ledger altogether (g) does not increase throughput substantially, but removing the overhead of executing transactions against the key-value store (h) again doubles throughput.

Tab. 3: Breakdown of IA-CCF features (f=1, dedicated cluster)

Variant	Throughput (tx/s)
(a) Full IA-CCF	47,841
(b) IA-CCF-NoReceipt	51,209
(c) + without checkpoints	51,288
(d) + small key-value store	53,759
(e) + without signed client requests	111,926
(f) + with MACs only	128,921
(g) + without ledger	131,959
(h) + with empty requests	299,321
HotStuff (with empty requests)	307,997
Pompē (with empty requests)	465,646

For context, we compare with two Byzantine consensus protocols with similar functionality to (h) above, HotStuff [62] and Pompē [66, 67]. HotStuff's throughput is 307,997 tx/s, but with higher latency (§6.2). By separating request ordering and consensus, Pompē achieves a throughput of 465,646 tx/s, also with worse latency (IA-CCF's 12 ms to Pompē's 73 ms). IA-CCF could utilize Pompē's techniques for increased throughput by sacrificing its two round-trip latency.

These breakdown results show that IA-CCF's overhead comes primarily from the cryptographic operations required for verifying client requests, followed by the transactional key-value store, rather than the consensus protocol or the mechanisms specific to providing individual accountability.

7   Related work

Permissioned ledgers. Many permissioned ledger systems [3, 4, 32, 52] rely on BFT consensus protocols to order transactions. Hyperledger Besu [32] and Quorum [52] use variants of PBFT [47, 55], which do not retain proof of a replica's operations, and therefore cannot assign blame. Diem [3] uses the DiemBFT [11] consensus protocol, which is based on HotStuff [62] and also lacks accountability features.

The IA-CCF prototype is built on top of CCF [54], an open source [44] distributed ledger framework deployed in the Azure cloud [43], which utilizes trusted execution environments (TEEs) [21, 35] to harden replicas. Russinovich et al. [54] describe CCF's programming model, receipts, governance, and replication protocols. CCF produces hardware attestation reports for the code running on each replica and adds them to the ledger. The ledger is signed by the CCF service and in the process binds CCF's public key to the code and hardware platform. While CCF enables auditing and can recover a ledger when all replicas crash, it relies on the security of TEEs, and its auditing does not guarantee individual accountability.

Byzantine consensus [17, 20, 37] distributes trust. Recent work on BFT protocols has focused on improving guarantees [5, 22, 46] or performance for particular use cases [57, 67]. SBFT [25] and HotStuff [62] scale to hundreds of replicas using threshold cryptography, which prevents blame assignment. For permissioned ledgers, scaling to many replicas without growing the consortium size does not improve trustworthiness, and consortia typically cannot grow arbitrarily.

Other work has explored misbehavior and its impact on Byzantine consensus. BFT2F [39] formalizes safety and liveness guarantees after more than f replicas are compromised. It provides PBFT's guarantees with up to f failures and provides *fork** consistency with up to $2f$ failures. For permissioned ledgers, *fork** consistency is not sufficient, because it is susceptible to double-spending attacks.

Depot [40] issues proofs-of-misbehavior after observing misbehavior, but it adopts eventual consistency, which is incompatible with permissioned ledgers. Pompē [67] prevents dishonest primaries from controlling the ordering of requests. It does not address scenarios in which there are more than f dishonest replicas though.

Accountability. PeerReview [27] ensures that distributed nodes remain accountable for their actions. As shown in §6.1, PeerReview incurs a high overhead when applied to a permissioned ledger. In contrast, IA-CCF introduces mechanisms specific to BFT state machine replication, such as a shared ledger with a Merkle tree, to improve both regular transaction execution and auditing.

Accountable virtual machines [26] carries out auditing through *spot checking* of checkpoints, but has the same performance overheads as PeerReview for ledgers. SNP [68] is a networking-specific implementation of accountability, offering provenance for routing decisions. Such specializations improve performance in particular domains, but are not directly applicable to permissioned ledgers.

BAR [1] and Prosecutor [65] incentivize replicas to act honestly by having honest replicas penalize misbehavior. This weaker model allows BAR to tolerate more than $1/3$ faulty replicas, while Prosecutor uses these incentives to improve performance. If these incentives fail [31], however, replicas share the blame.

Accountability with more than $f+1$ misbehaving replicas has been discussed before [14, 15, 28]. BFT Protocol Forensics [56] and Polygraph [19] propose a ledger auditing mechanism, but assume that fewer than $N-f$ replicas misbehave. They also do not support changing replica sets. ZLB [53] and Tendermint [14] support changes to the replica set but also assume that fewer than $N-f$ replicas misbehave.

8   Conclusions

In permissioned ledger systems, individual accountability is a strong disincentive for misbehavior. IA-CCF provides the evidence required to prove that $f+1$ or more replicas misbehaved when clients observe safety violations (even if all replicas fail). It offers strong consistency and security properties while providing state-of-the-art performance compared to existing ledger systems with weaker security guarantees. IA-CCF achieves this by integrating evidence collection for assigning blame with a novel ledger-based BFT consensus algorithm.

References

[1] Amitanand S Aiyer, Lorenzo Alvisi, Allen Clement, Mike Dahlin, Jean-Philippe Martin, and Carl Porth. BAR: Fault tolerance for cooperative services. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 45–58, 2005.

[2] Mohammad Alomari, Michael Cahill, Alan Fekete, and Uwe Rohm. The cost of serializability on platforms that use snapshot isolation. In *2008 IEEE 24th International Conference on Data Engineering*, pages 576–585. IEEE, 2008.

[3] Zachary Amsden, R Arora, S Bano, M Baudet, S Blackshear, A Bothra, G Cabrera, C Catalini, K Chalkias, E Cheng, et al. The Libra blockchain. *https://developers.diem.com/papers/the-diem-blockchain/2020-05-26.pdf*, 2019.

[4] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger Fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–15, 2018.

[5] Avi Asayag, Gad Cohen, Ido Grayevsky, Maya Leshkowitz, Ori Rottenstreich, Ronen Tamari, and David Yakira. A fair consensus protocol for transaction ordering. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, pages 55–65. IEEE, 2018.

[6] Diem association. An independent membership organization. https://diem.com/en-US/association/. Accessed: 2021-03-19.

[7] J Aythora, R Burke-Agüero, A Chamayou, S Clebsch, M Costa, N Earnshaw, L Ellis, P England, C Fournet, M Gaylor, et al. Multi-stakeholder media provenance management to counter synthetic media risks in news publishing. In *Proc. International Broadcasting Convention (IBC)*, 2020.

[8] American bar association. Arbitration. https://www.americanbar.org/groups/dispute_resolution/resources/DisputeResolutionProcesses/arbitration/. (Accessed on 03/27/2021).

[9] American bar association. How courts work. https://www.americanbar.org/groups/public_education/resources/law_related_education_network/how_courts_work/discovery/. (Accessed on 03/27/2021).

[10] Jeff Barr, Attila Narin, and Jinesh Varia. Building fault-tolerant applications on AWS. *Amazon Web Services*, pages 1–15, 2011.

[11] Mathieu Baudet, Avery Ching, Andrey Chursin, George Danezis, François Garillot, Zekun Li, Dahlia Malkhi, Oded Naor, Dmitri Perelman, and Alberto Sonnino. State machine replication in the Libra blockchain. *The Libra Assn., Tech. Rep*, 2019.

[12] Alysson Bessani, João Sousa, and Eduardo EP Alchieri. State machine replication for the masses with BFT-SMaRt. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362. IEEE, 2014.

[13] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 514–532. Springer, 2001.

[14] Ethan Buchman. *Tendermint: Byzantine fault tolerance in the age of blockchains*. PhD thesis, The University of Guelph, 2016.

[15] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437*, 2017.

[16] Carole Cadwalladr. Another huge data breach, another stony silence from facebook. https://www.theguardian.com/technology/2021/apr/11/another-huge-data-breach-another-stony-silence-from-facebook. (Accessed on 05/04/2021).

[17] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[18] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[19] Pierre Civit, Seth Gilbert, and Vincent Gramoli. Polygraph: Accountable Byzantine agreement. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 403–413. IEEE, 2021.

[20] Allen Clement, Mirco Marchetti, Edmund Wong, Lorenzo Alvisi, and Mike Dahlin. BFT: The time is now. In *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, pages 1–4, 2008.

[21] Victor Costan and Srinivas Devadas. Intel SGX explained. *Cryptology ePrint Archive*, 2016.

[22] Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. DBFT: Efficient leaderless Byzantine consensus and its application to blockchains. In *2018 IEEE*

17th International Symposium on Network Computing and Applications (NCA), pages 1–8. IEEE, 2018.

[23] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[24] Dan Goodin. Equifax website hack exposes data for ~143 million us consumers. https://arstechnica.com/information-technology/2017/09/equifax-website-hack-exposes-data-for-143-million-us-consumers/. (Accessed on 05/04/2021).

[25] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. SBFT: A scalable and decentralized trust infrastructure. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 568–580. IEEE, 2019.

[26] Andreas Haeberlen, Paarijaat Aditya, Rodrigo Rodrigues, and Peter Druschel. Accountable virtual machines. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[27] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. PeerReview: Practical accountability for distributed systems. *ACM SIGOPS operating systems review*, 41(6):175–188, 2007.

[28] Maurice Herlihy and Mark Moir. Blockchains and the logic of accountability: Keynote address. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 27–30, 2016.

[29] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[30] Chris Hourihan and Bryan Cline. A look back: US healthcare data breach trends. *Health Information Trust Alliance. Retrieved from https://hitrustalliance.net/content/uploads/2014/05/HITRUST-Report-US-Healthcare-Data-Breach-Trends. pdf*, 2012.

[31] Crypto Hustle. Krypton recovers from a new type of 51% network attack. http://cryptohustle.com/krypton-recovers-from-a-new-type-of-51-network-attack/. (Accessed on 12/06/2020).

[32] Hyperledger. Hyperledger Besu enterprise Ethereum client (Hyperledger Besu). https://besu.hyperledger.org/en/stable/. (Accessed on 12/06/2020).

[33] Hyperledger. The ordering service. https://hyperledger-fabric.readthedocs.io/en/release-2.2/orderer/ordering_service.html. (Accessed on 12/05/2020).

[34] IBM. we.trade | ibm. https://www.ibm.com/case-studies/wetrade-blockchain-fintech-trade-finance. (Accessed on 05/04/2021).

[35] David Kaplan, Jeremy Powell, and Tom Woller. AMD memory encryption. *White paper*, 2016.

[36] Eleftherios Kokoris Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 279–296, Austin, TX, 2016. USENIX Association.

[37] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative byzantine fault tolerance. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 45–58, 2007.

[38] Hemi Leibowitz, Ania M Piotrowska, George Danezis, and Amir Herzberg. No right to remain silent: Isolating malicious mixes. In *28th USENIX security symposium (USENIX security 19)*, pages 1841–1858, 2019.

[39] Jinyuan Li and David Maziéres. Beyond one-third faulty replicas in Byzantine fault tolerant systems. In *NSDI*, 2007.

[40] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. Depot: Cloud storage with minimal trust. *ACM Transactions on Computer Systems (TOCS)*, 29(4):1–38, 2011.

[41] SSL Library mbed TLS / PolarSSL. https://tls.mbed.org/. (Accessed on 12/09/2020).

[42] Ralph C Merkle. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*, pages 369–378. Springer, 1987.

[43] Microsoft. Confidential ledger - distributed ledger technology | Microsoft Azure. https://azure.microsoft.com/en-us/services/azure-confidential-ledger/. (Accessed on 02/04/2022).

[44] Microsoft. Microsoft/CCF: Confidential Consortium Framework. https://github.com/microsoft/ccf. (Accessed on 02/01/2022).

[45] Microsoft. Release ccf-0.13.2 · microsoft/CCF. https://github.com/microsoft/CCF/releases/tag/ccf-0.13.2. (Accessed on 01/13/2022).

[46] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of BFT protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 31–42, 2016.

[47] Henrique Moniz. The Istanbul BFT consensus algorithm. *arXiv preprint arXiv:2002.03613*, 2020.

[48] Takuya Nakaike, Qi Zhang, Yohei Ueda, Tatsushi Inagaki, and Moriyoshi Ohara. Hyperledger Fabric performance characterization and optimization using GoLevelDB benchmark. In *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–9. IEEE, 2020.

[49] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, 2014.

[50] Panavia. Introduction. https://www.panavia.de/company/introduction/. (Accessed on 05/04/2021).

[51] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, et al. Evercrypt: A fast, verified, cross-platform cryptographic provider. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 983–1002. IEEE, 2020.

[52] Quorum. A permissioned implementation of Ethereum supporting data privacy. https://github.com/ConsenSys/quorum. Accessed: 2020-11-27.

[53] Alejandro Ranchal-Pedrosa and Vincent Gramoli. ZLB: A blockchain to tolerate colluding majorities. *arXiv preprint arXiv:2007.10541*, 2020.

[54] Mark Russinovich, Edward Ashton, Christine Avanessians, Miguel Castro, Amaury Chamayou, Sylvan Clebsch, Manuel Costa, Cédric Fournet, Matthew Kerner, Sid Krishna, et al. CCF: A framework for building confidential verifiable replicated services. Technical report, Technical Report MSR-TR-2019-16, Microsoft, 2019.

[55] Roberto Saltini and David Hyland-Wood. IBFT 2.0: A safe and live variation of the IBFT blockchain consensus protocol for eventually synchronous networks. *arXiv preprint arXiv:1909.10194*, 2019.

[56] Peiyao Sheng, Gerui Wang, Kartik Nayak, Sreeram Kannan, and Pramod Viswanath. BFT protocol forensics. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1722–1743, 2021.

[57] Chrysoula Stathakopoulou, Tudor David, and Marko Vukolić. Mir-BFT: High-throughput BFT for blockchains. *arXiv preprint arXiv:1906.05552*, 2019.

[58] Michael J Steindorfer and Jurgen J Vinju. Fast and lean immutable multi-maps on the JVM based on heterogeneous hash-array mapped tries. *arXiv preprint arXiv:1608.01036*, 2016.

[59] Nick Szabo. The idea of smart contracts. *Nick Szabo's Papers and Concise Tutorials*, 6, 1997.

[60] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[61] Pieter Wuille. libsecp256k1. *URL: https://github.com/bitcoin/secp256k1*, 2018.

[62] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.

[63] Ted Yin and Dahlia Malkhi. GitHib - HotStuff. https://github.com/hot-stuff/libhotstuff/commit/df8328be09baeb81b7aaa037022eedaa7a416598. (Accessed on 04/15/2021).

[64] Aydan R Yumerefendi and Jeffrey S Chase. The role of accountability in dependable distributed systems. In *Proceedings of HotDep*, volume 5, pages 3–3. Citeseer, 2005.

[65] Gengrui Zhang and Hans-Arno Jacobsen. Prosecutor: An efficient BFT consensus algorithm with behavior-aware penalization against Byzantine attacks. In *Middleware*, 2021.

[66] Yunhao Zhang. GitHub - yhzhang0128/archipelago-hotstuff: the artifact for our OSDI'20 paper. https://github.com/yhzhang0128/archipelago-hotstuff. (Accessed on 04/27/2021).

[67] Yunhao Zhang, Srinath Setty, Qi Chen, Lidong Zhou, and Lorenzo Alvisi. Byzantine ordered consensus without Byzantine oligarchy. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 633–649, 2020.

[68] Wenchao Zhou, Qiong Fei, Arjun Narayan, Andreas Haeberlen, Boon Thau Loo, and Micah Sherr. Secure network provenance. In *Proceedings of the twenty-third ACM symposium on operating systems principles*, pages 295–310, 2011.

A Proof of L-PBFT linearizability

We present a correctness proof for L-PBFT. In particular, we show that *early execution* (Lemma 2) and the *nonce commitment* scheme (Lemma 3) are equivalent to their counterpart features in PBFT. In Thm. 1, we show linearizability of L-PBFT.

Lemma 1 (Rollback). *Any honest L-PBFT replica can roll back a suffix of the sequence of previously executed transaction batches.*

Proof. L-PBFT's state is distributed across several entities: a key-value store kv; a Merkle tree M; a ledger $\mathcal{L}$; a set of requests waiting to be ordered $\mathcal{T}$; a message store $\mathcal{M}$; and a nonce store $\mathcal{K}$. Therefore, to roll back a batch of transactions, it must be possible to roll back all of these entities.

Key-value store kv. The key-value store maintains a roll back transaction log. This enables transactions to be rolled back at a single transaction granularity. Thus, the last executed batch of transactions can be rolled back.

Merkle tree M. When a new node is added to L-PBFT's Merkle tree, it becomes the right-most leaf of the tree. The value of a node in the tree is never updated, and a node can only be deleted if it is the right-most node in the tree. Thus, during roll back, it is possible to remove the nodes from the right of the tree that represent the last batch of executed transactions (in reverse order).

Ledger $\mathcal{L}$. The ledger is represented by a file written to the disk by each replica. L-PBFT stores the index of all entries written to the ledger. To roll back the last executed batch, a L-PBFT replica truncates the ledger file to just before the first entry of the batch.

Transaction store $\mathcal{T}$. It is not necessary to undo changes to the transaction store. Transaction requests that are removed can be retransmitted by the client or other replicas if needed.

Message store $\mathcal{M}$, nonce store $\mathcal{K}$. All items in the transaction and nonce stores are indexed by sequence number and view. Since roll back occurs only during a view change, and each item is associated with a view, it is not necessary to modify the message and nonce stores, because honest replicas never send more than one item of a given type for the same sequence number and view.

Therefore, it is possible to roll back a suffix of the sequence of transaction batches executed by L-PBFT replicas. □

Lemma 2 (Early execution). *L-PBFT's early execution and PBFT execution agree on all committed transactions.*

Proof. In both PBFT and L-PBFT, the primary determines the order of request execution by ordering requests into batches and assigning numbers to batches in pre-prepare messages. In PBFT, requests are executed after commit and clients only accept results after transactions commit. In L-PBFT, requests are executed earlier, before the request even prepare, but the replicas only reply to clients after they prepare the requests and clients wait for matching replies from $N-f$ replicas. This ensures that they only obtain the transaction results after they commit as in PBFT.

As in PBFT, a faulty primary may cause requests for which pre-prepares are sent not to commit. L-PBFT deals with this case by rolling back early execution (see Lemma 1).
□

Lemma 3 (Nonce commitment). *The nonce commitment scheme is equivalent to replicas signing commit messages.*

Proof. L-PBFT, like PBFT, signs pre-prepare and prepare messages. Unlike PBFT, L-PBFT does not sign commit messages. Replicas sample a fresh random nonce for each pre-prepare or prepare message with sequence number s at view v, and add a hash of this nonce to the signed payloads. Later in the protocol, replicas include the nonce in the commit message, instead of an extra signature.

We show that this provides the same standard cryptographic security as the signature scheme (namely, resistance to existential forgery against chosen-message attacks) as long as the cryptographic hash function is second pre-image resistant on random inputs. Since the addition of a nonce to the signed payloads is injective, a forgery of a L-PBFT authenticator for a pre-prepare or prepare message yields a forgery against the signature scheme. A forgery of an authenticator for a commit message, i.e., a value with the same hash as a fresh random nonce that has not yet been revealed, is a second pre-image collision. □

Theorem 1. *L-PBFT is linearizable.*

Proof. L-PBFT changes the PBFT algorithm by adding early execution and the nonce commitment scheme. Lemmas 2 and 3 show that these preserve the behavior of PBFT.
□

B Proof of auditing correctness

First, we present the correctness proof for auditing without governance transactions and reconfiguration (§B.1). Then, we extend the proof to include governance transactions and reconfiguration (§B.2).

B.1 Correctness of auditing without reconfiguration

We begin with a description of terminology and notation. In §B.1.1 and Lemma 4, we then prove that, given a set of receipts, the auditor, with the help of the enforcer, can obtain a ledger package that is complete in relation to the receipts (or assign blame to $f+1$ misbehaving or slow replicas). A complete ledger package contains all evidence that is necessary for the auditor to assign blame to misbehaving replicas if the receipts reflect any linearizability violation. In §B.1.2 and Lemma 5, we show that, if a receipt does not appear correctly in a ledger package that is complete in relation to it, the auditor can assign blame to at least $f+1$ misbehaving replicas. In §B.1.3 and Lemma 6, using the previous lemmas, we first prove that the auditor can assign blame correctly if it is given a set of receipts that reflects a

serializability violation. Finally, Theorem 2 proves that, if a set of receipts reflects any linearizability violation, the auditor can assign blame to $f+1$ misbehaving or slow replicas.

Minimum ledger index. Each client transaction request includes a field that specifies the minimum ledger index that it can be executed at. Correct replicas do not order a transaction t at ledger index i, unless $i \geq m_i$ where m_i is the minimum index value of t. Correct clients set the minimum index of a transaction to at least M_i+1 where M_i is the largest value of the ledger index that they know of from the receipts that they have collected. The minimum index value is used to capture transaction dependencies efficiently and to reduce the amount of information that needs to be stored and transmitted to audit linearizability violations.

Ledger well-formedness and validity. A ledger fragment is *valid* if it can be produced by a sequence of correct primaries when there are at most f misbehaving replicas.

A ledger fragment is *well-formed* if either (i) it is valid, or (ii) it would be valid if not for the incorrect execution of one or more transactions, one or more incorrect checkpoint digests, or one or more invalid signatures or nonces.

A well-formed ledger matches the structural specifications of the L-PBFT protocol, i.e.,

- it specifies a serial ordering of transactions/entries, which respects their minimum ledger indices; and
- it includes evidence, and checkpoints at the required places.

A valid ledger is always well-formed, but a well-formed ledger can be invalid. A correct replica will never have a malformed ledger fragment, because replicas check the well-formedness of ledgers that they fetch. A correct replica may have an invalid ledger fragment. A ledger fragment can be well-formed but invalid only if there have at some point existed more than $N-f-1$ misbehaving replicas.

Notation. Given a receipt $\langle\langle t_j,i_j,o_j\rangle,x_j\rangle$, we denote $\langle t_j,i_j,o_j\rangle$ by tio_j. Unless explicitly defined otherwise, s_j refers to the sequence number in x_j of the receipt $\langle\text{tio}_j,x_j\rangle$.

We say that a replica has "signed a receipt" if its signature is recorded in the receipt in the pre-prepare/prepare signatures' fields (σ_p or in $\sum_s$).

Receipt validity. A receipt is *valid* if it is verifiable by Alg. 3.

Preparement evidence for a batch. The *preparement* evidence for a batch is $N-f$ signed pre-prepare/prepare messages for the batch, i.e., $\mathcal{P}$ in §3.

Checkpoint sequence numbers. Let $\langle\text{tio}_j,x_j\rangle$ be a valid receipt, d_{C_j} be the checkpoint digest in x_j, and C be the checkpoint interval. Anyone can calculate the sequence number at which the digest of the checkpoint is expected to be equal to d_{C_j} as follows: checkpoints are always taken at sequence numbers that are multiples of C and the digest in the receipt refers to the digest at the sequence number of the penultimate checkpoint transaction before s_j (except the first

C transactions, which have the digest at genesis). So given s_j, the sequence number with the corresponding checkpoint digest, s_{cp}, can be calculated as

$$s_{cp} = \begin{cases} 0 & \text{if } s_j < C \\ C(\lceil \frac{s_j}{C} \rceil - 2) & \text{otherwise.} \end{cases}$$

Note that the value of the digest itself is recorded in the last checkpoint transaction before s_j (except the first C transactions), i.e., the checkpoint transaction that follows the one at s_{cp}. That checkpoint transaction is at

$$\begin{cases} 0 & \text{if } s_j < C \\ s_{cp}+C & \text{otherwise.} \end{cases}$$

We assume that the genesis transaction gt is at sequence number 0.

Fetching checkpoints. Slow replicas can be brought up to date by fetching checkpoints and ledger fragments. When a correct replica fetches a checkpoint at sequence number s, it retrieves the ledger up to $s+C+P$. It first verifies the signatures in the evidence for the checkpoint transactions at s and $s+P$. Note that the replicas that signed the checkpoint transaction at s vouch for the validity of the ledger fragment between $s-C$ and s, whereas the replicas that signed the checkpoint transaction at $s+C$ vouch for the digest of the checkpoint at s.

A correct replica, then, verifies that the digest of the checkpoint that it fetched matches the value recorded at $s+C$. It also checks, for each checkpoint transaction at sequence number s' in the ledger, that the ledger's Merkle root at s' matches the root in the evidence for the transaction at s'. Finally, the replica replays the ledger fragment between $s+1$ and $s+C$.

As noted previously, a correct replica may have a well-formed ledger fragment that includes invalid signatures as replicas do not verify all signatures in the ledger fragments that they fetch. Therefore, when contacted for an audit, a correct replica never returns a ledger fragment that it fetched with a checkpoint at sequence number s, without including the checkpoint transaction at $s+C$ and the evidence for that transaction.

B.1.1 Obtaining the ledger

Ledger package. A *ledger package* from a replica consists of one to four components:

1. a ledger fragment $\mathcal{F}$ that contains entries that locally prepared at the replica;
2. an optional *suffix* $\mathcal{U}$ that contains entries that were preprepared atomically after a view-change but not yet prepared at the replica;
3. an optional *message box* $\mathcal{E}$ that contains some of the messages from the replica's message box $\mathcal{M}$; and
4. an optional *checkpoint cp*.

Complete ledger package. Let $\mathcal{R}$ be a set of valid receipts; s_{max} be the maximum sequence number in $\mathcal{R}$; s_{min} be the sequence number of the checkpoint whose digest is expected to equal the checkpoint digest in the receipt with the smallest sequence number in $\mathcal{R}$ (s_{min} can be calculated as described in the previous section); v_{min} and v_{max} be the minimum and maximum view numbers in the receipts in $\mathcal{R}$, respectively.

A ledger package is *complete* in relation to $\mathcal{R}$ if all of the following are true:

- $\mathcal{F} + \mathcal{U}$ is well-formed;
- if $s_{min} = 0$, cp contains the checkpoint at genesis (empty); otherwise, the digest of cp is equal to the one in the second checkpoint transaction in $\mathcal{F} + \mathcal{U}$;
- $\mathcal{F}$ includes at least one set of *view-change* and *new-view* messages for a view less than or equal to $v_{min} + 1$ (v_{min} requirement), and one set of *view-change* and *new-view* messages for a view greater than or equal to v_{max} (v_{max} requirement);
- All signatures in $\mathcal{F} + \mathcal{U}$ and $\mathcal{E}$ are valid.

and one of the following is true:

- $\mathcal{F}$ includes entries between s_{min} and $s_{max} + P$;
- $\mathcal{F}$ includes entries between s_{min} and $s_{max} + c$ where $c \in [0, P)$. $\mathcal{E}$ contains $P - c$ valid preparement evidence for entries from $s_{max} - c$ to s_{max}; or
- $\mathcal{F}$ includes entries between s_{min} and $e = \max(s_{min}, s_{max} - c)$ where $c \in [1, P]$. $\mathcal{E}$ contains valid preparement evidence for entries from $\max(s_{min}, e - P)$ to e. The suffix $\mathcal{U}$ contains entries between $e + 1$ and s_{max} that are preprepared but not prepared in some view $v' \geq v_{max}$ and $\mathcal{E}$ contains preparement evidence from a view $< v'$ for entries between $e + 1$ and s_{max}.

Lemma 4 (Obtaining a complete ledger package). *Given a set of valid receipts $\mathcal{R}$, an auditor can either obtain a ledger package that is complete in relation to $\mathcal{R}$, or assign blame to at least $f + 1$ misbehaving or slow replicas.*

Proof. Select from the receipts in $\mathcal{R}$, the receipts with the highest view number v_{max}. Then, from those receipts select the receipts with the highest sequence number. Finally, among those, let R_{vmax} be the receipt with the highest index number. (We assume there is no tie; otherwise, the auditor assigns blame to the replicas that signed both tied receipts.)

The enforcer asks all replicas that signed R_{vmax} for a ledger package that is complete in relation to $\mathcal{R}$. We assume that correct replicas or members respond to the enforcer before the agreed deadline. Once the enforcer has responses from $f + 1$ replicas, it relays the responses to the auditor; otherwise at the deadline, the enforcer assigns blame to at least $f + 1$ misbehaving or slow replicas.

We show that a correct replica can either respond with: a ledger package that is complete in relation to $\mathcal{R}$ or a ledger package with which the auditor can assign blame to $f + 1$ misbehaving replicas. Therefore, after checking $f + 1$ responses, the auditor either finds a complete ledger package, or assigns blame to $f + 1$ misbehaving replicas.

Note that a correct replica that is contacted by the enforcer can always satisfy the first three conditions of completeness: (1) correct replicas always maintain well-formed ledgers and they record/can recalculate checkpoints; (2) the v_{min} requirement can always trivially be satisfied by including the set of *view-change* and *new-view* messages for view 0 in $\mathcal{F}$. In practice, for efficiency, correct replicas would satisfy this requirement by including the set of *view-change* and *new-view* messages for some view v', where v' is the latest possible in $[0, v_{min} + 1]$; and (3) since the replicas that are asked are the replicas that signed R_{vmax}, they must have *view-change* and *new-view* messages for view v_{max}. Therefore, any replica that returns a ledger package that violates any of the first three conditions can be assigned blame.

The fourth condition of completeness requires that all signatures and the matching nonces in the ledger package are correct. Let $\langle \mathcal{F}, \mathcal{U}, \mathcal{E}, cp \rangle$ be a ledger package returned by a replica. If $\mathcal{U}$ or $\mathcal{E}$ contains a message or transaction with an invalid signature, the auditor can assign blame to the replica. $\mathcal{E}$ contains messages from the replica's message box and $\mathcal{U}$ contains batches that pre-prepared at the replica. A correct replica never considers a message or pre-prepares a batch that includes an invalid signature. Otherwise, let s_w be a sequence number where there is a transaction or message with an invalid signature. The auditor can look for the first checkpoint transaction that follows s_w that has no invalid signatures in its evidence. If one exists, the auditor can assing blame to all $N - f$ replicas that signed that checkpoint transaction. If no such checkpoint transaction exists, the auditor can assign blame to the responding replica, since a correct replica never returns a ledger fragment that it has fetched with a checkpoint without including the committed checkpoint transaction that records that checkpoint's digest. So given a ledger package from a replica, the auditor can always verify all signatures and nonces in the package or assign blame to the responding replica or $N - f$ misbehaving replicas. So below, for brevity, we can assume that the ledger package that a replica returns has no invalid signatures or nonces.

Additionally, for a correct replica that is contacted by the enforcer, one of the following must hold:

- **The correct replica has locally prepared entries up to at least s_{max}:** In this case, the replica can form a complete ledger package that includes either:
    - (i) a well-formed ledger fragment $\mathcal{F}$ that contains entries from s_{min} to $s_{max} + P$; or
    - (ii) a well-formed $\mathcal{F}$ that contains entries from s_{min} to $s_{max} + c$ where $c \in [0, P)$, and $\mathcal{E}$ that contains $P - c$ valid preparement evidence for entries from $s_{max} - c$ to s_{max}.

- **The correct replica has not locally prepared entries up to $s_{\max}$ and it has locally prepared entries up to $e = s_{\max} - c$ where $c \geq 1$:** In this case, (1) a correct replica can include entries between $s_{\min}$ and e in a well-formed ledger fragment $\mathcal{F}$, and it can include the necessary preparement evidence in $\mathcal{E}$ (if $s_{\min} \leq e$); and (2) if the replica has any batches that it has preprepared but not prepared due to a view-change, it can include the related *view-change* and *new-view* messages in $\mathcal{F}$ and the batches in $\mathcal{U}$. Let p be the last sequence number for which there is a batch in $\mathcal{F} + \mathcal{U}$. If $p > e$, the correct replica can include the preparement evidence for entries between $e+1$ and p in $\mathcal{E}$ as well. A correct replica can form a ledger package as described above. If $p \geq s_{\max}$, the ledger package is complete, and the replica can return it. Otherwise, $p < s_{\max}$. Let $R_{s_{\max}}$ be the receipt in $\mathcal{R}$ with the largest sequence number $s_{\max}$ and let $v_{s_{\max}}$ be the view number in $R_{s_{\max}}$. Note that $v_{s_{\max}} \leq v_{\max}$ by definition, and in the correct replicas' ledger, there must exist at least one set of *view-change* and *new-view* messages for a view $v' > v_{s_{\max}}$ such that none of the *view-change* messages include a pre-prepare message for any batch at $s_{\max}$. The correct replica can return a ledger package that contains these *view-change* and *new-view* messages. The auditor can use the returned ledger package to assign blame to the intersection of replicas that signed $R_{s_{\max}}$ and that sent the set of *view-change* messages for v', as these replicas have prepared a batch at $s_{\max}$ but did not report it during the view change.

Thus, for each of the $f+1$ responses, either the response is complete in relation to $\mathcal{R}$, or the auditor can assign blame to the misbehaving responder, or at least $f+1$ misbehaving replicas. □

By definition of completeness, if a ledger package is complete in relation to a set of valid receipts $\mathcal{R}$, it is complete in relation to any subset of $\mathcal{R}$.

Finding preparement evidence. For a batch at s_r, the auditor can find the preparement evidence for the batch as follows:

- if $\mathcal{F}$ contains an entry at $s_r + P$, it is collected from there;
- if $\mathcal{F}$ contains the entry at s_r but not at $s_r + P$, it is collected from $\mathcal{E}$; and
- if $\mathcal{F}$ does not contain an entry at s_r but $\mathcal{U}$ contains an entry at s_r, it is also collected from $\mathcal{E}$, albeit it is for the same batch from a prior view.

B.1.2 Incompatibility

Let $R = \langle \text{tio}_r, x_r \rangle$ be a valid receipt at sequence number s_r. Let $\langle \mathcal{F}, \mathcal{U}, \mathcal{E}, cp \rangle$ be a ledger package that is complete in relation to R. Let B_l be the batch that is at s_r in $\mathcal{F} + \mathcal{U}$. R is *incompatible* with B_l if any of the following hold:

- t_r does not appear in B_l;
- it does not appear in the i_rth position; or
- o_r is different.

Lemma 5 (Receipt-ledger incompatibility). *Let $R = \langle \text{tio}_r, x_r \rangle$ be a valid transaction receipt for sequence number s_r. Let $\langle \mathcal{F}, \mathcal{U}, \mathcal{E}, cp \rangle$ be a ledger package that is complete in relation to R. Let B_l be the batch in the package at s_r. If R is incompatible with B_l, the auditor can assign blame to at least $f+1$ misbehaving replicas.*

Proof. The auditor can calculate the set of replicas that signed B_l using the preparement evidence that can be found as described above. These replicas are called $\mathcal{E}_l$.

Let $\mathcal{E}_r$ be the set of replicas that have signed the receipt. Let v_r be the view number in the receipt and v_l be the view number in the preparement evidence of B_l.

- $v_r = v_l$: Correct replicas never sign pre-prepare or prepare messages for different batches in the same view. Therefore, the auditor can assign blame to the replicas in the intersection of $\mathcal{E}_r$ and $\mathcal{E}_l$, and $|\mathcal{E}_r \cap \mathcal{E}_l| \geq f+1$.
- $v_l > v_r$: Correct replicas include the pre-prepare messages for the last P prepared batches in their *view-change* messages until the batches commit or a different batch is prepared at the sequence number. A correct primary always re-preprepares the latest batch that it finds in the set of $N - f$ *view-change* messages that it receives. Thus, there exists at least one view $v_c \in [v_r + 1, v_l]$ where zero of the $N - f$ *view-change* messages for v_c contain a pre-prepare message for the batch at sequence number s_r that is referenced in R. The ledger package is complete in relation to R, so $\mathcal{F}$ includes at least one set of *view-change* and *new-view* messages for a view less than or equal to $v_r + 1$ (the $v_{\min}$ requirement). It must also include the set of *view-change* and *new-view* messages for v_c as $v_l \geq v_c \geq v_r + 1$.
  Let $\mathcal{E}_c$ be the set of replicas that have sent the *view-change* messages to the primary for view v_c. The auditor can assign blame to the replicas that are in the intersection of $\mathcal{E}_r$ and $\mathcal{E}_c$ and $|\mathcal{E}_r \cap \mathcal{E}_c| \geq f+1$.
- $v_l < v_r$: There exists at least one view $v_c \in [v_l + 1, v_r]$ where zero of the $N - f$ *view-change* messages for v_c contains a pre-prepare message for the batch at sequence number s_r that is referenced in R. The ledger package is complete in relation to R so $\mathcal{F}$ includes at least one set of *view-change* and *new-view* messages for a view greater than or equal to v_r, so it must include the set of *view-change* and *new-view* messages for v_c as $v_l + 1 \leq v_c \leq v_r$ (the $v_{\max}$ requirement). Similar to previous case afterwards.

□

B.1.3 Violations

Ordering receipts. Given a set of valid receipts, the auditor can order them lexicographically based on the corresponding (sequence number, index number, view number) tuples. (We can assume that there is no tie; otherwise, the auditor assigns blame to the replicas that signed both tied receipts.) We say that a receipt R_1 is *earlier/later* than a receipt R_2, if it

is ordered before/after R_2 with this scheme, respectively. For example, the earliest receipt in a set of valid receipts is the one with the lowest view number, among those with the lowest index number, among those with the lowest sequence number.

Lemma 6 (Serializability violations). *Let $\mathcal{R} = \{(tio_0, x_0), ..., (tio_k, x_k))\}$ be a set of valid receipts that violates serializability. Then, the auditor can assign blame to at least $f+1$ misbehaving or slow replicas.*

Proof. First, the auditor can obtain a ledger package $\langle \mathcal{F}, \mathcal{U}, cp, \mathcal{E} \rangle$ that is complete in relation to $\mathcal{R}$; otherwise, it can assign blame to at least $f+1$ misbehaving or slow replicas by Lemma 4. Note that, as the ledger package is complete in relation to $\mathcal{R}$, it is complete in relation to any receipt $R_j \in \mathcal{R}$.

Since the receipts in $\mathcal{R}$ violate serializability, no serial execution of $t_0, ..., t_k$ can produce $io_0, ..., io_k$. $\mathcal{F} + \mathcal{U}$ is well-formed, so there are two options for its validity:

Valid ledger. $\mathcal{F} + \mathcal{U}$ is a valid ledger, so every transaction in it is ordered and executed serially. However, the receipts in $\mathcal{R}$ violate serializability. Therefore, there must exist at least one receipt $\langle tio_w, x_w \rangle \in R$ that is incompatible with the batch at s_w in $\mathcal{F} + \mathcal{U}$. By Lemma 5, the auditor can assign blame to at least $f+1$ misbehaving replicas.

Invalid ledger. $\mathcal{F} + \mathcal{U}$ is a well-formed but invalid ledger. So there exists at least one transaction t_w (which does not have to be in $\mathcal{R}$) that was executed incorrectly in some batch s_w, or one checkpoint that was created incorrectly.

The auditor can order $\mathcal{R}$ as described above. Let R_e be the earliest receipt in $\mathcal{R}$. Let d_{C_0} be the checkpoint digest in R_e. Let s_{C_0} be the sequence number with the expected checkpoint digest d_{C_0}, calculated by the auditor using s_e and the checkpoint interval C as previously described. If $s_{C_O} = 0$, but the digest in R_e is not equal to the digest in the genesis transaction, the auditor can assign blame to all replicas that signed R_e. Otherwise, the ledger package is complete with respect to R_e, and $\mathcal{F} + \mathcal{U}$ is thus well-formed, so: (i) the entry at s_{C_0} in $\mathcal{F} + \mathcal{U}$ is a checkpoint transaction; and (ii) the checkpoint transaction in $s_{C_O} + C$ exists as $s_{C_0} < s_{C_O} + C < s_e$ and contains the digest of cp. If the digest of cp in the ledger package is not d_{C_0}, the auditor can assign blame to the replicas that signed both the checkpoint transaction at $s_{C_O} + C$ and R_e. The digest in that checkpoint transaction is for the previous checkpoint and the batches before the previous checkpoint have already committed since $C > P$.

Otherwise, the auditor replays the ledger starting from the checkpoint transaction at s_{C_0}, creating checkpoints at checkpoint sequence numbers. Doing so, the auditor either obtains $\langle t_w, i_w, o_a \rangle \neq \langle t_w, i_w, o_w \rangle$ or finds that an incorrect checkpoint digest is recorded at s_w. In either case, the auditor can assign blame to all replicas that signed for the batch at s_w. $\square$

Theorem 2 (Linearizability violations). *Let $\mathcal{R}$ be a set of receipts that violate linearizability. Then, the auditor can*

assign blame to at least $f+1$ misbehaving or slow replicas.

Proof. If the receipts also violate serializability, the auditor can assign blame to at least $f + 1$ misbehaving or slow replicas by Lemma 6.

Otherwise, since the receipts violate linearizability but not serializability, the ordering of the transactions in $\mathcal{R}$ must violate the real-time ordering of the transactions. So there exists at least two transactions, t_a and t_b, in $\mathcal{R}$ such that the receipt for tio_a was received by the client before t_b was sent, but $i_a \geq i_b$. t_b was sent after $\langle tio_a, x_a \rangle$ was received, so a correct client sets the minimum index l of tio_b to at least $i_a + 1$. Since $i_b \leq i_a$, the auditor can assign blame to all replicas who have sent the receipt for tio_b. $\square$

B.2 Correctness of auditing with reconfiguration

In this section, we first summarize how reconfiguration happens, introduce new terminology, and update prior terminology. Then, in Lemma 7, we prove that, if the auditor detects a fork in governance, it can assign blame to $f+1$ misbehaving replicas. In §B.2.1, we update the prior discussion on obtaining a complete ledger package. In §B.2.2 and Lemma 9, we prove that, if a receipt and the corresponding batch in a ledger package are prepared in different configurations, the auditor can assign blame to $f + 1$ misbehaving replicas. In §B.2.3, using Lemma 9, we update the prior lemma about incompatibility. Finally, §B.2.4 updates the prior proofs on violations, and in Theorem 3, we prove the correctness of auditing in the complete IA-CCF ledger system.

Summary of reconfiguration. A correct primary ends the batch it is working on once it executes a governance transaction. Therefore, each batch includes at most one governance transaction and i_g in a receipt refers to the last governance transaction executed before the transaction in the receipt. The final vote transaction that is necessary to pass a referendum triggers the configuration change. $2P$ *end-of-config* batches follow the final vote before the configuration change. The governance sub-ledger consists of batches and evidence for all governance transactions. It also includes, for each configuration, the Pth and $2P^{\text{th}}$ *end-of-config* batches, which commit the final vote transaction that triggers reconfiguration and the Pth *end-of-config* batch respectively. The Pth *end-of-config* batch links to the final vote transaction, because its pre-prepare message includes the Merkle root of the batch that includes the final vote transaction.

Updates to well-formedness and validity. A ledger fragment is *valid* if it can be produced by a sequence of correct primaries in a sequence of configurations where in each configuration there are at most f failures.

In addition to the previous structural specifications, governance changes are serialized and include the required *end-of-config* and *start-of-config* messages.

Note that correct replicas check the validity of the governance sub-ledger fragments that they fetch, so their

governance sub-ledgers are valid, in addition to well-formed.

Configuration number. The configuration number of a configuration C is the distance that it is from the configuration at the genesis. The genesis has configuration number 0. A configuration that follows the genesis configuration has number 1 and so on.

Supporting governance chain of a receipt. Every receipt R includes the index of the latest governance transaction. A correct client makes sure that it has a matching chain of valid governance transaction receipts for each receipt that it has. This includes the receipts for all governance transactions from the genesis up to the latest governance transaction, and the receipt for the Pth *end-of-config* batch for each configuration change. The supporting governance chain of a receipt R is the sequence of governance-related receipts that starts from the genesis transaction receipt and ends with the Pth *end-of-config* batch receipt before the configuration that signed R takes effect.

A supporting governance chain of a receipt matches a governance sub-ledger if each receipt in the chain is compatible with the governance sub-ledger. (For *end-of-config* batches, compatibility considers committed Merkle roots as well.) Similarly, a supporting governance chain can be a prefix of a governance sub-ledger.

Updates to receipt validity. A receipt is *valid* if it is verifiable by Alg. 3, and it is attached a valid supporting governance chain.

Updates to calculating checkpoint sequence numbers. If a sequence number that is multiple of the checkpoint interval C falls into an *end-of-config/start-of-config* sequence, checkpointing is skipped. A checkpoint is taken at the beginning of each new configuration, and the digest of the first checkpoint in a configuration is included in the first checkpoint transaction, as opposed to the one that follows (this is similar to genesis).

Let $\langle tio_j, x_j \rangle$ be a valid receipt and s_{fv} be sequence number of the final vote transaction for the last configuration change in the supporting governance chain of the receipt. The first checkpoint of the configuration that prepared the receipt is expected at $s_{fcp} = s_{fv} + 2P + 1$. (Except the genesis configuration, for which $s_{fcp} = 0$.)

So given s_j, the sequence number s_{cp} of the checkpoint whose digest is in x_j can be calculated with

$$s_{cp} = \begin{cases} s_{fcp} & \text{if } s_j < s_{fcp} + C \\ C\left(\left\lceil \frac{s_j - s_{fcp}}{C} \right\rceil - 2\right) & \text{otherwise.} \end{cases}$$

Updates to fetching checkpoints. Following a configuration change, a correct new replica fetches the checkpoint at the penultimate checkpoint sequence number s' in the previous configuration (or the first checkpoint sequence number if there is only one). It also retrieves the full ledger. It

replays the ledger from s' before creating a checkpoint at the beginning of the configuration.

Equivalence of Pth *end-of-config* batches. Two P^{th} *end-of-config* batches are equivalent if they:

(i) are at the same index and sequence number; and
(ii) are preceded by the same valid governance sub-ledger (their pre-prepares include the same committed Merkle root).

Two receipts for P^{th} *end-of-config* batches are equivalent if the batches specified in them are equivalent.

Governance fork. There is a fork in governance if there is a fork in the governance sub-ledger. That is, there are at least two P^{th} *end-of-config* batches for the same configuration number that belong in valid governance sub-ledgers, but that are not equivalent.

We say that there is a fork between two valid supporting governance chains if there are receipts for two P^{th} *end-of-config* batches for the same configuration number that are not equivalent.

We say that there is a fork between a valid supporting governance chain and a valid governance sub-ledger, if for the same configuration number, the Pth *end-of-config* batch specified by the receipt in the chain is not equivalent to the Pth *end-of-config* batch in the sub-ledger.

Lemma 7 (Governance fork). *If there is a fork in governance, the auditor can assign blame to at least $f + 1$ misbehaving replicas.*

Proof. If there is a fork in governance, there are at least two P^{th} *end-of-config* batches for the same configuration number that are not equivalent, namely P_1 and P_2.

A correct replica only prepares a Pth *end-of-config* batch at sequence number s once the final vote transaction that passes the referendum is committed at sequence number $s - P$. Thus, all governance transactions preceding it are committed too. This final vote transaction triggers the configuration change.

So the auditor can assign blame to the replicas that prepared both P_1 and P_2, because a correct replica that prepares one will never prepare another non-equivalent P^{th} *end-of-config* batch in the same configuration number. $\square$

Longest supporting governance chain. Let $\mathcal{R}$ be a set of valid receipts. If there is a fork between the supporting governance chains of the receipts in $\mathcal{R}$, the auditor can assign blame to at least $f + 1$ misbehaving replicas by Lemma 7. So the auditor can always obtain a *longest supporting governance chain* for the receipts in $\mathcal{R}$. This chain is the union of all supporting chains for receipts in $\mathcal{R}$.

Onwards, we assume that, given any set of valid receipts, the supporting governance chains are fork-free with each other and that there is a longest supporting governance chain;

otherwise, the auditor can assign blame to $f+1$ misbehaving replicas by Lemma 7.

Transaction receipts. Onwards, we assume that a receipt is for a transaction and not for *end-of-config/start-of-config* batches. If the receipts for *end-of-config/start-of-config* indicate a fork in governance, misbehaving replicas can be blamed using Lemma 7; otherwise, the *end-of-config/start-of-config* batches do not have any usage and do not affect the key-value store, so do not affect linearizability.

B.2.1 Updates to obtaining the ledger

Updated ledger package. A ledger package includes an additional required field:

- the committed governance sub-ledger $\mathcal{N}$ of the replica.

Updated definition of completeness. Let $\mathcal{R}$ be a set of valid receipts. Define $s_{max}, v_{min}, v_{max}$ as previously. Calculate s_{min} using the receipt with the smallest configuration number, among those with the smallest sequence number in $\mathcal{R}$. Let n_{gmax} be the longest supporting governance chain in $\mathcal{R}$.

A ledger package is *complete* in relation to $\mathcal{R}$ if, in addition to the prior conditions about well-formedness, length, and v_{min}/v_{max} requirements:

- n_{gmax} is a prefix of $\mathcal{N}$ (i.e. the package is obtained from a replica in a configuration which is equal to or succeeds all configurations in $\mathcal{R}$);
- $\mathcal{N}$ is valid; and
- $\mathcal{N}$ matches $\mathcal{F}$.

  The condition for the checkpoint cp is updated as follows:

- if s_{min} is calculated as the first checkpoint transaction in a configuration (or zero), the digest of cp is equal to the one in the checkpoint transaction at s_{min}; otherwise, the digest of cp is equal to the one in the second checkpoint transaction in $\mathcal{F} + \mathcal{U}$.

Lemma 8 (Obtaining a complete ledger package with reconfiguration). *Given a set of valid receipts $\mathcal{R}$, an auditor can either obtain a ledger package that is complete in relation to $\mathcal{R}$, or assign blame to at least $f+1$ misbehaving or slow replicas.*

Proof. As mentioned before, we assume that there is no fork between the supporting governance chains of the receipts in $\mathcal{R}$. Let R_{gmax} be the receipt with the highest index number, among those with the highest sequence number, among those with the highest view number, among those with the longest supporting governance chain in $\mathcal{R}$. Let n_{gmax} be the supporting governance chain of R_{gmax}.

We assume that there is a reliable mechanism to look up the most recent system configuration. Using this mechanism, the auditor looks up the most recent committed governance sub-ledger and the set of replicas that signed the first checkpoint transaction of the most recent configuration. If there is a fork

between n_{gmax} and the governance sub-ledger that is looked-up, the auditor can assign blame to at least $f+1$ misbehaving replicas by Lemma 7; otherwise, the auditor checks whether the sub-ledger that is looked up is longer than n_{gmax}. If so, the enforcer asks all the replicas that signed the first checkpoint transaction of the most recent configuration for a ledger package; otherwise, the replicas that have signed R_{gmax} are asked.

As in Lemma 4, the enforcer asks replicas for a ledger package that is complete in relation to $\mathcal{R}$. At the deadline, the enforcer relays the responses to the auditor. There are at least $f+1$ responses, or the enforcer can assign blame to $f+1$ misbehaving or slow replicas.

As before, we show that a correct replica can either respond with: a ledger package that is complete in relation to $\mathcal{R}$, or a ledger package with which the auditor can assign blame to $f+1$ misbehaving replicas.

First, note that a correct replica that is contacted by the enforcer can always satisfy the updated completeness conditions (related to $\mathcal{N}$), because the replica is part of the most recent configuration and the conditions all pertain to keeping a valid governance sub-ledger. Of the conditions described previously, the well-formedness and v_{min} conditions can be satisfied, and invalid signatures in the package can be handled, just as in Lemma 4. Since the replicas that are asked are not necessarily the replicas that signed the receipt with the highest view in $\mathcal{R}$, it is possible that they cannot satisfy the v_{max} requirement even if they are correct.

So, for a correct replica that is contacted by the enforcer one of the following must hold:

- **The replica cannot satisfy the v_{max} requirement:** Let R_{vmax} be the latest receipt when the receipts are ordered lexycographically by (view number, configuration number, sequence number, index number). Let n_{vmax} be the supporting governance chain of R_{vmax}. If there is a fork between n_{vmax} and the committed sub-ledger $\mathcal{N}$ of the replica, the replica can return its governance sub-ledger and the auditor can assign blame to at least $f+1$ misbehaving replicas by Lemma 7. Otherwise, n_{vmax} must be a prefix of $\mathcal{N}$ since the enforcer asks replicas from the most recent configuration. There are two possibilities for the relationship between n_{vmax} and $\mathcal{N}$:

  1. $\mathcal{N} = n_{vmax}$. So $R_{gmax} = R_{vmax}$. Therefore, the correct replica signed R_{vmax}. Any correct replica that signed R_{vmax} has the *view-change* and *new-view* messages for v_{max}, so this case is a contradiction.
  2. $\mathcal{N}$ is longer than n_{vmax}. Let P_{vmax+1} be the Pth *end-of-config* batch that ends R_{vmax}'s configuration C. Since the replica is correct and cannot satisfy the v_{max} requirement, P_{vmax+1} must be prepared in a view $< v_{max}$. Any correct replica that prepared P_{vmax+1} must have committed a final vote transaction that triggers the configuration change in their ledger in a view less than v_{max}. Since correct replicas never reset their ledger by

more than P sequence numbers, they do not pre-prepare any batch with view $v_{\max}$ in C. So, the auditor can assign blame to the intersection of replicas that signed $R_{v\max}$ and prepared $P_{v\max+1}$.

- **The replica can satisfy the $v_{\max}$ requirement:** If, additionally, the replica has prepared (or pre-prepared with view changes) batches up to at least $s_{\max}$, it can return a ledger package that is complete in relation to $\mathcal{R}$ just as in Lemma 4.

  Otherwise, let $R_{s\max}$ be the receipt with the largest sequence number $s_{\max}$. Let $n_{s\max}$ be the supporting governance chain of $R_{s\max}$. If there is a fork between $n_{s\max}$ and the replica's $\mathcal{N}$, the replica can return $\mathcal{N}$ and the auditor can assign blame to at least $f+1$ misbehaving replicas by Lemma 7. Otherwise, $n_{s\max}$ must be a prefix of $\mathcal{N}$ since the replicas asked by the enforcer are from the most recent configuration. Again, there are two possibilities:

  1. $\mathcal{N}$ **is longer than** $n_{s\max}$**:** Let $P_{s\max+1}$ be the Pth *end-of-config* batch that ends $R_{s\max}$'s configuration. Since the replica is correct and cannot satisfy the $s_{\max}$ requirement, $P_{s\max+1}$ must be prepared at a sequence number less than $s_{\max}$. Any correct replica that prepared $P_{s\max+1}$ must have committed a final vote transaction that triggers the configuration change at latest at sequence number $s_{\max} - (P + 1)$. Since a correct replica never resets its ledger by more than P sequence numbers, the auditor can assign blame to the replicas that signed both $R_{s\max}$ and prepared $P_{s\max+1}$.
  2. $\mathcal{N} = n_{s\max}$**:** The group of replicas asked by the enforcer are from the same configuration that signed $R_{s\max}$, which is the most recent configuration. Since the replica is correct and from the most recent configuration $v_{s\max} \le v_{\max}$ by definition. In $\mathcal{F}$, as before, there must exist at least one set of *view-change* and *new-view* messages for a view $v' > v_{s\max}$ such that none of the *view-change* messages includes a pre-prepare for any batch at $s_{\max}$. Note that the configuration of the replicas that have sent these *view-change* messages must be the same as the configuration that signed the receipt, as that is the most recent configuration in the system. So just as in Lemma 4, the auditor can assign blame to the replicas that signed both $R_{s\max}$ and that sent the set of *view-change* messages for v'.

So, for each of the $f+1$ responses, either the response is complete in relation to $\mathcal{R}$, or the auditor can assign blame to the responder, or at least $f+1$ misbehaving replicas. □

B.2.2 Mismatching configurations

Lemma 9 (Receipt-ledger configuration mismatch)**.** *Let* $R = \langle tio_r, x_r \rangle$ *be a valid receipt that was produced in a configuration* C_r*. Let* B_l *be the batch that is at* s_r *in a ledger package that is complete in relation to R. Let* C_l *be the config-*

uration of the replicas that signed B_l. If $C_r \ne C_l$, the auditor can assign blame to at least $f+1$ misbehaving replicas.

Proof. Since R is a valid receipt, it has a valid supporting governance chain. Since the ledger package is complete, it includes a valid governance sub-ledger $\mathcal{N}$ that leads to C_l, which is fork-free with the supporting governance chain of R. One of the following must hold:

- $C_r < C_l$**:** C_r **precedes** C_l**:** Let P_{r+1} be the Pth *end-of-config* batch that ends the configuration C_r. This batch and its evidence is included in $\mathcal{N}$. Since the package is complete, $\mathcal{N}$ is consistent with the ledger fragment in the package. Since that ledger fragment is well-formed and B_l is at s_r, P_{r+1} is at the latest at sequence number $s_r - (P+1)$. Any replica that prepared P_{r+1} must have committed a final vote transaction that triggers the configuration change at the latest at sequence number $s_r - (2P+1)$. A correct replica that has prepared a batch at s_r in C_r never resets its ledger to earlier than $s_r - P$ even with view changes. So the auditor can assign blame to the replicas that both prepared P_{r+1} and signed R.
- $C_r > C_l$**:** C_r **succeeds** C_l**:** We show that this case is impossible given that R is valid, and there is no fork between its supporting governance chain and $\mathcal{N}$. Since the ledger package is complete in relation to R, $\mathcal{N}$ includes the Pth *end-of-config* batch leading to C_r and it matches the well-formed ledger fragment in the package. Since B_l is at s_r, that batch can at earliest be at sequence number $s_r + P$. So there cannot be a valid receipt produced in C_r at s_r.

□

B.2.3 Updates to incompatibility

Lemma 10 (Receipt-ledger incompatibility with reconfiguration)**.** *Let* $R = \langle tio_r, x_r \rangle$ *be a valid transaction receipt at sequence number* s_r*. Let* $\langle \mathcal{F}, \mathcal{U}, \mathcal{E}, cp, \mathcal{N} \rangle$ *be a ledger package that is complete in relation to R. Let* B_l *be the batch in the package at* s_r*. If R is incompatible with* B_l*, the auditor can assign blame to at least* $f+1$ *misbehaving replicas.*

Proof. Define $\mathcal{E}_l, \mathcal{E}_r, v_l, v_r$ as in Lemma 5. Note that we can assume that both the receipt and B_l are prepared by the same configuration C; if not, the auditor can assign blame to $f+1$ misbehaving replicas by Lemma 9.

- $v_r = v_l$**:** Same as Lemma 5.
- $v_l > v_r$**:** Calculate $\mathcal{E}_c$ as described in Lemma 5. If the replicas in $\mathcal{E}_c$ are also from the configuration C, the auditor can assign blame just as in Lemma 5; otherwise, if the replicas in $\mathcal{E}_c$ are from a preceding configuration, the first checkpoint transaction of C is at the latest at sequence number $s_r - (P+1)$ since B_l is prepared by C and $\mathcal{F} + \mathcal{U}$ is well-formed. Furthermore, that checkpoint transaction is prepared in a view $v' > v_r$. A correct replica never signs the receipt at s_r in a view v_r and then resets its ledger by more than P sequence numbers while view changing to v'.

So, the auditor can assign blame to the replicas that signed both that checkpoint transaction and the receipt.

- $v_l < v_r$: Calculate $\mathcal{E}_c$ as described in Lemma 5. If the replicas in $\mathcal{E}_c$ are also from the configuration $\mathcal{C}$, the auditor can assign blame just as in Lemma 5; otherwise the replicas in $\mathcal{E}_c$ are from a configuration that succeeds $\mathcal{C}$. In this case, the Pth *end-of-config* batch that ends the configuration $\mathcal{C}$ is at the earliest at sequence number $s_r + P$, since B_l is prepared by $\mathcal{C}$ and $\mathcal{F} + \mathcal{U}$ is well-formed. Furthermore, that batch is prepared in a view $v' < v_r$. A correct replica that prepares that P^{th} *end-of-config* batch commits to the configuration change; it never resets its ledger to earlier than s_r and signs R. So, the auditor can assign blame to the replicas that signed both that *end-of-config* batch and the receipt.

$\square$

B.2.4 Updates to violations

Lemma 11 (Serializability violations with reconfiguration).
Let $\mathcal{R} = \{(tio_0, x_0), ..., (tio_k, x_k))\}$ be a set of receipts that violates serializability. Then, the auditor can assign blame to at least $f + 1$ misbehaving or slow replicas.

Proof. First, the auditor can obtain a ledger package $\langle \mathcal{F}, \mathcal{U}, cp, \mathcal{E}, \mathcal{N} \rangle$ that is complete in relation to $\mathcal{R}$; otherwise, IA-CCF can assign blame to at least $f + 1$ misbehaving or slow replicas by Lemma 8.

Just as in Lemma 6, since the receipts in $\mathcal{R}$ violate serializability, no serial execution of $t_0, ..., t_k$ can produce $io_0, ..., io_k$. $\mathcal{F}$ is well-formed, so there are two options for its validity:

Valid ledger. Similar to Lemma 6. By Lemma 10, the auditor can assign blame to at least $f + 1$ misbehaving replicas.

Invalid ledger. Assume that receipts are ordered lexicographically based on the corresponding (sequence number, configuration number, index number, view number) tuples. (We can assume that there is no tie; otherwise the auditor can assign blame to the replicas that signed both tied receipts.)

Let R_e be the earliest receipt in the ordered $\mathcal{R}$. Let d_{C_0} be the digest in R_e. Let s_{C_0} be the sequence number with the expected checkpoint digest d_{C_0}. s_{C_0} can be calculated by the auditor using s_e, the checkpoint interval C, and the supporting governance chain. (Note that s_{C_0} is equal to $s_{\min}$ that is calculated while obtaining the ledger.)

We can assume that the batch at s_e is prepared by the same configuration that sent the receipt; otherwise the auditor can assign blame to $f + 1$ misbehaving replicas by Lemma 9. We also know that the supporting governance chain of R_e matches $\mathcal{F} + \mathcal{U}$ and that $\mathcal{F} + \mathcal{U}$ is well-formed. So, the checkpoint transactions at s_{C_0} (and $s_{C_0} + C$ if it exists) are prepared by the same configuration as R_e by definition of s_{C_0}. So, if the digest at s_{C_0} is not d_{C_0}, the auditor can assign blame to $f + 1$ misbehaving replicas similar to Lemma 6.

Since the supporting governance chains of all receipts match the ledger fragment by definition of completeness, the auditor can determine the correct stored procedures for each transaction to replay the ledger as in Lemma 6. $\square$

Theorem 3 (Linearizability violations with reconfiguration).
Let $\mathcal{R}$ be a set of receipts that violate linearizability. Then, the auditor can assign blame to at least $f + 1$ misbehaving or slow replicas.

Proof. If the receipts also violate serializability, the auditor can assign blame to at least $f + 1$ misbehaving or slow replicas by Lemma 11; otherwise, the minimum ledger index argument in the proof of Theorem 2 holds. $\square$

DispersedLedger: High-Throughput Byzantine Consensus on Variable Bandwidth Networks

Lei Yang[1], Seo Jin Park[1], Mohammad Alizadeh[1], Sreeram Kannan[2], David Tse[3]

[1]*MIT CSAIL* [2]*University of Washington* [3]*Stanford University*

Abstract

The success of blockchains has sparked interest in large-scale deployments of Byzantine fault tolerant (BFT) consensus protocols over wide area networks. A central feature of such networks is variable communication bandwidth across nodes and across time. We present DispersedLedger, an asynchronous BFT protocol that provides near-optimal throughput in the presence of such variable network bandwidth. The core idea of DispersedLedger is to enable nodes to propose, order, and agree on blocks of transactions without having to download their full content. By enabling nodes to agree on an ordered log of blocks, with a guarantee that each block is available within the network and unmalleable, DispersedLedger decouples bandwidth-intensive block downloads at different nodes, allowing each to make progress at its own pace. We build a full system prototype and evaluate it on real-world and emulated networks. Our results on a geo-distributed wide-area deployment across the Internet shows that DispersedLedger achieves 2× better throughput and 74% reduction in latency compared to HoneyBadger, the state-of-the-art asynchronous protocol.

1 Introduction

State machine replication (SMR) is a foundational task for building fault-tolerant distributed systems [25]. SMR enables a set of nodes to agree on and execute a replicated log of commands (or *transactions*). With the success of cryptocurrencies and blockchains, *Byzantine fault-tolerant SMR (BFT)* protocols, which tolerate arbitrary behavior from adversarial nodes, have attracted considerable interest in recent years [2, 5, 7, 15, 31, 39]. The deployment environment for these protocols differs greatly from standard SMR use cases. BFT implementations in blockchain applications must operate over wide-area networks (WAN), among possibly hundreds to thousands of nodes [2, 18, 31].

Large-scale WAN environments present new challenges for BFT protocols compared to traditional SMR deployments across a few nodes in datacenter. In particular, WANs are subject to *variability* in network bandwidth, both across different nodes and across time. While BFT protocols are secure in the presence of network variability, their performance can suffer greatly.

To understand the problem, let us consider the high-level structure of existing BFT protocols. BFT protocols operate in epochs, consisting of two distinct phases: (i) a *broadcast* phase, in which one or all of the nodes (depending on whether the protocol is leader-based [1, 39] or leaderless [17, 31]) broadcast a *block* (batch of transactions) to the others; (ii) an *agreement* phase, in which the nodes vote for blocks to append to the log, reaching a verifiable agreement (e.g., in the form of a quorum certificate [11]). From a communication standpoint, the broadcast phase is bandwidth-intensive while the agreement phase typically comprises of multiple rounds of short messages that do not require much bandwidth but are latency-sensitive.

Bandwidth variability hurts the performance of BFT protocols due to *stragglers*. In each epoch, the protocol cannot proceed until a super-majority of nodes have downloaded the blocks and voted in the agreement phase. Specifically, a BFT protocol on $N = 3f + 1$ nodes (tolerant to f faults) requires votes from at least $2f + 1$ nodes to make progress [11]. Therefore, the throughput of the protocol is gated by the $(f+1)^{th}$ slowest node in each epoch. The implication is that low-bandwidth nodes (which take a long time to download blocks) hold up the high-bandwidth nodes, preventing them from utilizing their bandwidth efficiently. Stragglers plague even asynchronous BFT protocols [31], which aim to track actual network performance (without making timing assumptions), but still require a super-majority to download and vote for blocks in each epoch. We show that this lowers the throughput of these protocols well below the average capacity of the network on real WANs.

In this paper, we present DispersedLedger, a new approach to build BFT protocols that significantly improves performance in the presence of bandwidth variability. The key idea behind this approach is to decompose consensus into two steps, one of which is not bandwidth intensive and the other is. First, nodes agree on an ordered log of *commitments*, where each commitment is a small digest of a block (e.g., a Merkle root [30]). This step requires significantly less bandwidth than downloading full blocks. Later, each node downloads the blocks in the agreed-upon order and executes the transactions to update its state machine. The principal advantage of this approach is that each node can download blocks at its own pace. Importantly, slow nodes do not impede
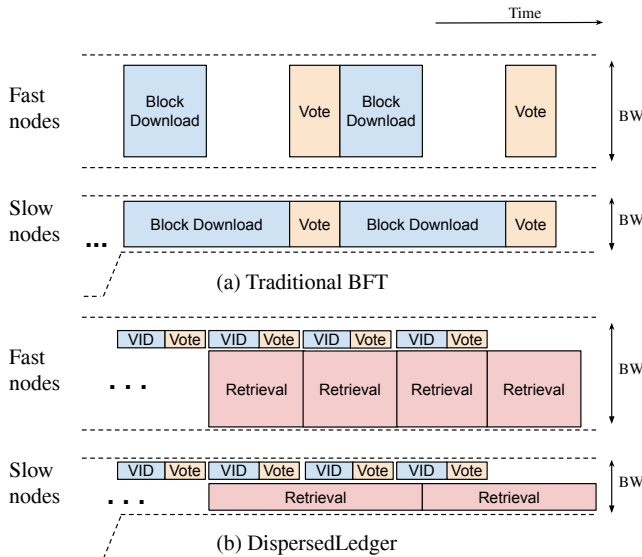
Figure 1: Impact of bandwidth variability on overall performance. Bcast: broadcast, Agmt: agreement. Fast nodes currently have a high bandwidth, while slow nodes currently have low bandwidth. (a) In traditional BFT protocols, the speed of consensus is always limited by the slow nodes since they take a long time to download the blocks. (b) DispersedLedger allows each node to download blocks at its own pace as permitted by its bandwidth.

the progress of fast nodes as long as they have a minimal amount of bandwidth needed to participate in the first step.

The key to realizing this idea is to guarantee the *data availability* of blocks. When a node accepts a commitment into the log, it must know that the block referred to by this commitment *is* available in the network and can be downloaded at a later time by any node in the network. Otherwise, an attacker can put a commitment of an unavailable block into the log, thus halting the system. To solve this problem, our proposal relies on *Verifiable Information Dispersal (VID)* [10]. VID uses erasure codes to store data across N nodes, such that it can be retrieved later despite Byzantine behavior. Prior BFT protocols like HoneyBadger [31] have used VID as a communication-efficient broadcast mechanism [10], but we use it to guarantee data availability. Specifically, unlike HoneyBadger, nodes in DispersedLedger do not wait to download blocks to vote for them. They vote as soon as they observe that a block has been *dispersed*, and the next epoch can begin immediately once there is agreement that dispersal has completed. This enables slow nodes to participate in the latest epoch, even if they fall behind on block downloads (retrieval). Such nodes can catch up on retrievals when their bandwidth improves. Figure 1 shows the structure of DispersedLedger, contrasting it to traditional BFT protocols.

Enabling nodes to participate in a consensus protocol with minimal bandwidth has applications beyond improving performance on temporally fluctuating bandwidth links. It also creates the possibility of a network with two types of

nodes: high-bandwidth nodes and low-bandwidth nodes. All nodes participate in agreeing on the ordered log of commitments, but only the high-bandwidth nodes retrieve all blocks. Network participants can choose what mode to use at any time. For example, a node running on a mobile device can operate in the low-bandwidth mode when connected to a cellular network, and switch to high-bandwidth mode on WiFi to catch up on block retrievals. All nodes, both high-bandwidth and low-bandwidth, contribute to the network's security. Our approach is also a natural way to shard a blockchain [27], where different nodes only retrieve blocks in their own shard.

We make the following contributions:

- We propose a new asynchronous VID protocol, AVID-M (§3). Compared to the current state-of-the-art, AVID-M achieves 1–2 orders of magnitudes better communication cost when operating on small blocks (hundreds of KBs to several MBs) and clusters of more than a few servers.
- We design DispersedLedger (§4), an asynchronous BFT protocol based on HoneyBadger [31] with two major improvements: (i) It decomposes consensus into data availability agreement and block retrieval, allowing nodes to download blocks asynchronously and fully utilize their bandwidth (§4.2). (ii) It provides a new solution to the censorship problem [31] that has existed in such BFT protocols since [4] (§4.3). Unlike HoneyBadger, where up to f correct blocks can get dropped every epoch, our solution guarantees that *every* correct block is delivered (and executed). The technique is applicable to similarly-constructed protocols, and can improve throughput and achieve censorship resilience without advanced cryptography [31].
- We address several practical concerns (§4.5): (i) how to prevent block retrieval traffic from slowing down dispersal traffic, which could reduce system throughput; (ii) how to prevent constantly-slow nodes from falling arbitrarily behind the rest of the network; (iii) how to avoid invalid "spam" transactions, now that nodes may not always have the up-to-date system state to filter them out.
- We implement DispersedLedger in 8,000 lines of Go (§5) and evaluate it in multiple settings (§6), including two global testbeds on AWS and Vultr, and controlled network emulations. DispersedLedger achieves a throughput of 36 MB/s when running at 16 cities across the world, and a latency of 800 ms that is stable across a wide range of load. Compared to HoneyBadger, DispersedLedger has 105% higher throughput and 74% lower latency.

2 Background and Related Work

2.1 The BFT Problem

DispersedLedger solves the problem of Byzantine-fault-tolerant state machine replication (BFT) [25]. In general, BFT assumes a server-client model, where N servers maintain N replicas of a state machine. At most f servers are Byzantine and may behave arbitrarily. Clients may submit transactions to a correct server to update or read the state machine. A

BFT protocol must ensure that the state machine is replicated across all correct servers despite the existence of Byzantine servers. Usually, this is achieved by delivering a consistent, total-ordered log of transactions to all servers (nodes) [31]. Formally, a BFT protocol provides the following properties:

- **Agreement**: If a correct server executes a transaction m, then all correct servers eventually execute m.
- **Total Order**: If correct servers p and q both execute transactions m_1 and m_2, then p executes m_1 before m_2 if and only if q executes m_1 before m_2.
- **Validity**: If a correct client submits a transaction m to a correct server, then all correct servers eventually execute m.[1]

There are multiple trust models between BFT servers and the clients. In this paper, we assume a model used for *consortium blockchains* [2,3,6,40], where servers and clients belong to organizations. Clients send their transactions through the servers hosted by their organization and trust these servers. Many emerging applications of BFT like supply chain tracing [14], medical data management [26], and cross-border transaction clearance [22] fall into this model.

2.2 Verifiable Information Dispersal

DispersedLedger relies on verifiable information dispersal (VID). VID resembles a distributed storage, where clients can *disperse* blocks (data files) across servers such that they are available for later *retrieval*. We provide a formal definition of VID in §3.1. The problem of *information dispersal* was first proposed in [37], where an erasure code was applied to efficiently store a block across N servers without duplicating it N times. [19] extended the idea to the BFT setting under the asynchrony network assumption. However, it did not consider Byzantine clients; these are malicious clients which try to cause two retrievals to return different blocks. *Verifiable* information dispersal (VID) was first proposed in [10], and solved this inconsistency problem. However, [10] requires that *every* node downloads the *full* block during dispersal, so it is no more efficient than broadcasting. The solution was later improved by AVID-FP [21], which requires each node to only download an $O(1/N)$ fraction of the dispersed data by utilizing fingerprinted cross-checksums [21]. However, because every message in AVID-FP is accompanied by the cross-checksum, the protocol provides low communication cost only when the dispersed data block is much larger than the cross-checksum (about $37N$ bytes). This makes AVID-FP unsuitable for small data blocks and clusters of more than a few nodes. In §3, we revisit this problem and propose AVID-M, a new asynchronous VID protocol that greatly reduces the per-message overhead: from $37N$ bytes to the size of a single hash (32 bytes), independent of the cluster size N, making the protocol efficient for small blocks and large clusters.

---

[1] Some recent BFT protocols provide a weaker version of validity, which guarantees execution of a transaction m only after being sent to *all* correct servers. This is referred to by different names: "censorship resilience" in HoneyBadger, and "fairness" in [8,9].

2.3 Asynchronous BFT protocols

A distributed algorithm has to make certain assumptions on the network it runs on. DispersedLedger makes the weakest assumption: *asynchrony* [28], where messages can be arbitrarily delayed but not dropped. A famous impossibility result [16] shows there cannot exist a deterministic BFT protocol under this assumption. With randomization, protocols can tolerate up to f Byzantine servers out of a total of $3f+1$ [24]. DispersedLedger achieves this bound.

Until recently [31], asynchronous BFT protocols have been costly for clusters of even moderate sizes because they have a communication cost of at least $O(N^2)$ [8]. HoneyBadger [31] is the first asynchronous BFT protocol to achieve $O(N)$ communication cost per bit of committed transaction (assuming batching of transactions). The main structure of HoneyBadger is inspired by [4], and it in turn inspires the design of other protocols including BEAT [15] and Aleph [17]. In these protocols, all N nodes broadcast their proposed blocks in each epoch, which triggers N parallel Binary Byzantine Agreement (BA) instances to agree on a subset of blocks to commit. [10] showed that VID can be used as an efficient construction of reliable broadcast, by invoking retrieval immediately after dispersal. HoneyBadger and subsequent protocols use this construction as a blackbox. BEAT [15] explores multiple trade-offs in HoneyBadger and proposes a series of protocols based on the same structure. One protocol, BEAT3, also includes a VID subcomponent. However, BEAT3 is designed to achieve BFT *storage*, which resembles a distributed key-value store.

2.4 Security Model

Before proceeding, we summarize our security model. We make the following assumptions:

- The network is asynchronous (§2.3).
- The system consists of a fixed set of N nodes (servers). A subset of at most f nodes are Byzantine, and $N \geq 3f+1$. N and f are protocol parameters, and are public knowledge.
- Messages are authenticated using public key cryptography. The public keys are public knowledge.

3 AVID-M: An Efficient VID Protocol

3.1 Problem Statement

VID provides the following two primitives: `Disperse`(B), which a client invokes to disperse block B, and `Retrieve`, which a client invokes to retrieve block B. Clients invoke the `Disperse` and `Retrieve` primitives against a particular *instance* of VID, where each VID instance is in charge of dispersing a different block. Multiple instances of VID may run concurrently and independently. To distinguish between these instances, clients and servers tag all messages of each VID instance with a unique ID for that instance. For each instance of VID, each server triggers a `Complete` event to indicate that the dispersal has completed.

A VID protocol must provide the following properties [10] for each instance of VID:

- **Termination**: If a correct client invokes `Disperse`(B) and no other client invokes `Disperse` on the same instance, then all correct servers eventually `Complete` the dispersal.
- **Agreement**: If some correct server has `Completed` the dispersal, then all correct servers eventually `Complete` the dispersal.
- **Availability**: If a correct server has `Completed` the dispersal, and a correct client invokes `Retrieve`, it eventually reconstructs some block B'.
- **Correctness**: If a correct server has `Completed` the dispersal, then correct clients always reconstruct the *same* block B' by invoking `Retrieve`. Also, if a correct client initiated the dispersal by invoking `Disperse`(B) and no other client invokes `Disperse` on the same instance, then $B = B'$.

3.2 Overview of AVID-M

At a high level, a VID protocol works by encoding the dispersed block using an erasure code and storing the encoded *chunks* across the servers. A server knows a dispersal has completed when it hears from enough peers that they have received their chunks. To retrieve a dispersed block, a client can query the servers to obtain the chunks and decode the block. Here, one key problem is verifying the correctness of encoding. Without verification, a malicious client may distribute *inconsistent* chunks that have more than one decoding result depending on which subset of chunks are used for decoding, violating the Correctness property. As mentioned in §2.2, AVID [10] and AVID-FP solve this problem by requiring servers to download the chunks or fingerprints of the chunks from all correct peers and examine them during dispersal. While this eliminates the possibility of inconsistent encoding, the extra data download required limits the scalability of these protocols.

More specifically, while AVID-FP [21] can achieve optimal communication complexity as the block size $|B|$ goes to infinity, its overhead for practical values of $|B|$ and N (number of servers) can be quite high. This is because every message in AVID-FP is accompanied by a fingerprinted cross-checksum [21], which is $N\lambda + (N-2f)\gamma$ in size. Here, λ, γ are security parameters, and we use $\lambda = 32$ bytes, $\gamma = 16$ bytes as suggested by [21]. The key factor that limits the scalability of AVID-FP is that the size of the cross-checksum is proportional to N. Combined with the fact that a node receives $O(N)$ messages during dispersal, the overhead caused by cross-checksum increases quadratically as N increases. Fig. 2 shows the impact of this overhead. At $N > 40$, $|B| = 100$ KB, every node needs to download more than the *full size* of the block being dispersed.

We develop a new VID protocol for the asynchronous network model, Asynchronous Verifiable Information Dispersal with Merkle-tree (AVID-M). AVID-M is based on one key observation: as long as clients can independently verify the encoding during *retrieval*, the servers do not need to do the verification during dispersal. In AVID-M, a client invoking `Disperse`(B) commits to the set of (possibly inconsistent)
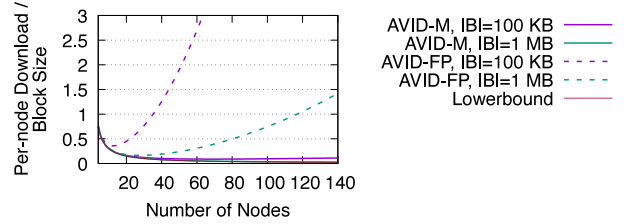

Figure 2: *Per-node* communication cost during dispersal of AVID-M and AVID-FP normalized over the size of the dispersed block. At $N = 128$ (the biggest cluster size in our evaluation), every node in AVID-M downloads as much as $1/32$ of a block, while a node in AVID-FP downloads $1.2\times$ the size of the full block.

chunks using a short, constant-sized commitment H. Then the server-side protocol simply agrees on H and guarantees enough chunks that match H are stored by correct servers. This can be done by transmitting only H in the messages, compared to the $O(N)$-sized cross-checksums in AVID-FP. During retrieval, a client verifies that the block it decodes produces the same commitment H when re-encoded.

Since AVID-M's per-message overhead is a small constant (32 bytes), it can scale to many nodes without requiring a large block size. In fact, AVID-M achieves a per-node communication cost of $O(|B|/N + \lambda N)$, much lower than AVID-FP's $O(|B|/N + \lambda N^2 + \gamma N^2)$. Fig. 2 compares AVID-M with AVID-FP. At $|B| = 1$ MB, AVID-M is close to the theoretical lower-bound[2] even at $N > 100$, while AVID-FP stops to provide any bandwidth saving (compared to every server downloading full blocks) after $N > 120$. Finally, we note that both AVID-M and AVID-FP rely on the security of the hash. So with the same hash size λ, AVID-M is no less secure than AVID-FP.

3.3 AVID-M Protocol

The **Dispersal algorithm** is formally defined in Fig. 3. A client initiates a dispersal by encoding the block B using an $(N-2f, N)$-erasure code and constructing a Merkle tree [30] out of the encoded chunks. The root r of the Merkle tree is a secure summary of the array of the chunks. The client sends one chunk to each server along with the Merkle root r and a Merkle *proof* that proves the chunk belongs to root r. Servers then need to make sure at least $N-2f$ chunks under the same Merkle root are stored at *correct* servers for retrieval. To do that, servers exchange a round of `GotChunk`(r) messages to announce the reception of the chunk under root r. When $N - f$ servers have announced `GotChunk`(r), they know at least $N - 2f$ correct servers have got the chunk under the same root r, so they exchange a round of `Ready`(r) messages to collectively `Complete` the dispersal.

---

[2]Each node has to download at least $\frac{1}{N-2f}$-fraction of the dispersed data. This is to prevent a specific attack: a malicious client sends chunks to all f malicious servers plus $N-2f$ honest servers. For now the malicious servers do not deviate from the protocol, so the protocol must terminate (otherwise it loses liveness). Then the malicious servers do not release the chunks, so the original data must be constructed from the $N-2f$ chunks held by honest servers, so each honest server must receive an $\frac{1}{N-2f}$-fraction share.

Disperse(B) **invoker**
1. Encode the input block B using an $(N-2f,N)$-erasure code, which results in N chunks, $C_1,C_2,...,C_N$.
2. Form a Merkle tree with all chunks $C_1,C_2,...,C_N$, and calculate the Merkle tree root, r.
3. Send Chunk(r,C_i,P_i) to the i-th server. Here P_i is the Merkle proof showing C_i is the i-th chunk under root r.

Disperse(B) **handler for the i-th server**
• Upon receiving Chunk(r,C_i,P_i) from a client:
1. Check if C_i is the i-th chunk under root r by verifying the proof P_i. If not, ignore the message.
2. Set MyChunk $= C_i$, MyProof $= P_i$, MyRoot $= r$ (all initially unset).
3. Broadcast GotChunk(r) if it has not sent a GotChunk message before.
• Upon receiving GotChunk(r) from the j-th server:
1. Increment ShareCount[r] (initially 0).
2. If ShareCount[r] $\geq N-f$, broadcast Ready(r).
• Upon receiving Ready(r) from the j-th server:
1. Increment ReadyCount[r] (initittally 0).
2. If ReadyCount[r] $\geq f+1$, broadcast Ready(r).
3. If ReadyCount[r] $\geq 2f+1$, set ChunkRoot $= r$. Dispersal is Complete.

Figure 3: Algorithm for Disperse(B). Servers ignore duplicate messages (same sender and same type). When broadcasting, servers also send the message to themselves.

The **Retrieval algorithm** is formally defined in Fig 4. A client begins retrieval by requesting chunks for the block from all servers. Servers respond by providing the chunk, the Merkle root r, and the Merkle proof proving that the chunk belongs to the tree with root r. Upon collecting $N-2f$ different chunks with the same root, the client can decode and obtain a block B'. However, the client must ensure that other retrieving clients also obtain B' no matter which subset of $N-2f$ chunks they use – letting clients perform this check is a key idea of AVID-M. To do that, the client re-encodes B', constructs a Merkle tree out of the resulting chunks, and verifies that the root is the same as r. If not, the client returns an error string as the retrieved content.

The AVID-M protocol described in this section provides the four properties mentioned in §3.1. We provide a proof sketch for each property, and point to Appendix B for complete proofs.

Termination (Theorem B.2). A correct client sends correctly encoded chunks to all servers with root r. The $N-f$ correct servers will broadcast GotChunk(r) upon getting their chunk. All correct servers will receive the $N-f$ GotChunk(r) and send out Ready(r), so all correct servers will receive at least $N-f$ Ready(r). Because $N-f > 2f+1$, all correct servers will Complete.

Agreement (Theorem B.4). A server Completes after receiving $2f+1$ Ready(r), of which $f+1$ must come from correct servers. So all correct servers will receive at least

Retrieve **invoker**
• Broadcast RequestChunk to all servers.
• Upon getting ReturnChunk(r, C_i, P_i) from the i-th server:
1. Check if C_i is the i-th chunk under root r by verifying the proof P_i. If not, ignore the messsage.
2. Store the chunk C_i with the root r.
• Upon collecting $N-2f$ or more chunks with the same root r:
1. Decode using any $N-2f$ chunks with root r to get a block B'. Set ChunkRoot $= r$ (initially unset).
2. Encode the block B' using the same erasure code to get chunks $C_1',C_2',...,C_N'$.
3. Compute the Merkle root r' of $C_1',C_2',...,C_N'$.
4. Check if $r' =$ ChunkRoot. If so, return B'. Otherwise, return string "BAD_UPLOADER".

Retrieve **handler for the i-th server**
• Upon receiving RequestChunk, respond with message ReturnChunk(ChunkRoot, MyChunk, MyProof) if MyRoot $=$ ChunkRoot. Defer responding if dispersal is not Complete or any variable here is unset.

Figure 4: Algorithm for Retrieve. Clients ignore duplicate messages (same sender and same type).

$f+1$ Ready(r). This will drive all of them to send Ready(r). Eventually every correct server will receive $N-f$ Ready(r), which is enough to Complete ($N-f > 2f+1$).

Availability (Theorem B.6). To retrieve, a client must collect $N-2f$ chunks with the *same* root. This requires that at least $N-2f$ correct servers have a chunk for the same root. Now suppose that a correct server Completes when receiving $2f+1$ Ready(r). When this happens, at least one correct server has sent Ready(r). We prove that this implies that at least $N-2f$ correct servers must have sent GotChunk(r) (Lemma B.1),i.e., they have received the chunk. Assume the contrary. Then there will be less than $N-f$ GotChunk(r). Now a correct server only sends Ready(r) if it either receives (i) at least $N-f$ GotChunk(r), or (ii) at least $f+1$ Ready(r). Neither is possible (see Lemma B.1).

All correct servers agree on the same root upon Complete by setting ChunkRoot to the same value (Lemma B.5). To see why, notice that each server will only send one GotChunk per instance. If correct servers Complete with 2 (or more) ChunkRoots, then at least $N-f$ servers must have sent GotChunk for each of these roots. But $2(N-f) > N+f$, hence at least one correct server must have sent GotChunk for two different roots, which is not possible.

Correctness (Theorem B.9). First, note that two correct clients finishing Retrieve will set ChunkRoot to be the same, i.e., they will decode from chunks under the same Merkle root r (Lemma B.5). However, we don't know if two different subsets of chunks under r would decode to the same block, because a malicious client could disperse arbitrary data as chunks. To ensure consistency of Retrieve across
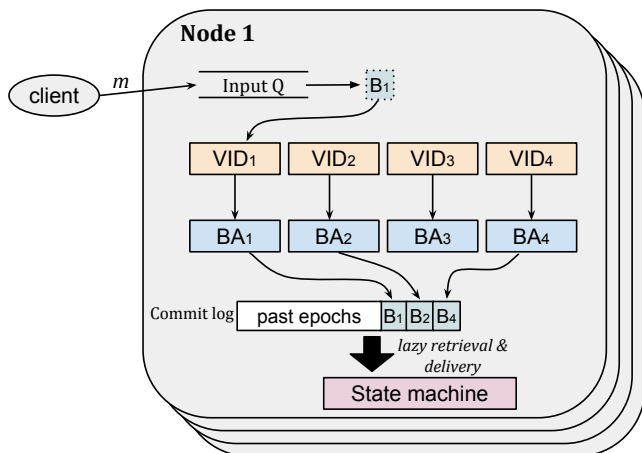
Figure 5: DispersedLedger architecture with $N = 4$. During this single epoch, 4 VIDs are initiated, one for each node, and three blocks B_1, B_2 and B_4 are committed.

different correct clients, every correct client re-encodes the decoded block B', calculates the Merkle root r' of the encoding result, and compares r' with the root r. There are two possibilities: (i) Some correct client gets $r' = r$. Then r corresponds to the chunks given by the *correct* encoding of B', so every correct client decoding from any subset of blocks under r will also get B' and $r' = r$. (ii) No correct client gets $r' = r$, i.e, all of them get $r' \neq r$. In this case, they all deliver the fixed error string. In either case, all correct clients return the same data (Lemma B.8).

4 DispersedLedger Design

4.1 Overview

DispersedLedger is a modification of HoneyBadger [31], a state-of-the-art asynchronous BFT protocol. HoneyBadger runs in epochs, where each epoch commits between $N - f$ to N blocks (at most 1 block from each node). As shown in Fig. 5, transactions submitted by clients are stored in each node's input queue. At the beginning of each epoch, every node creates a block from transactions in its input queue, and proposes it to be committed to the log in the current epoch. Once committed, all transactions in the block will eventually be retrieved and delivered to the state machine for execution.

DispersedLedger has two key differences with HoneyBadger. First, unlike HoneyBadger, a node in DispersedLedger does not broadcast its proposed block; instead, it *disperses* the proposed block among the entire cluster using AVID-M (which we will refer to as VID from here on). As shown in Fig. 5, there are N instances of VID in every epoch, one for each node. DispersedLedger then relies on N instances of Binary Agreement (BA, details below) [32] to reach a consensus on which proposed blocks have been successfully dispersed and thus should be committed in the current epoch. Once committed, a block can be retrieved by nodes lazily at any time (concurrently with future block proposals and dispersals). The asynchronous retrieval of blocks allows each node to

Figure 6: Algorithm for single-epoch DispersedLedger.

adapt to temporal network bandwidth variations by adjusting the rate it retrieves blocks without slowing down other nodes.

In HoneyBadger, up to f correct blocks can be dropped in every epoch (§4.3). This wastes bandwidth and can lead to censorship where blocks from certain nodes are always dropped [31]. DispersedLedger's second innovation is a new method, called inter-node linking, that guarantees every correct block is committed.

DispersedLedger uses an existing BA protocol [32] that completes in $O(1)$ time (parallel rounds) with $O(N\lambda)$ per-node communication cost, where λ is the security parameter. In BA, each node provides a binary $\texttt{Input}(\{0,1\})$ as input to the protocol, and may get an $\texttt{Output}(\{0,1\})$ event indicating the result of the BA instance. Formally, a BA protocol has the following properties:

- **Termination**: If all correct nodes invoke $\texttt{Input}$, then every correct node eventually gets an $\texttt{Output}$.
- **Agreement**: If any correct node gets $\texttt{Output}(b)$ ($b \in \{0,1\}$), then every correct node eventually gets $\texttt{Output}(b)$.
- **Validity**: If any correct node gets $\texttt{Output}(b)$ ($b \in \{0,1\}$), then at least one correct node has invoked $\texttt{Input}(b)$.

4.2 Single Epoch Protocol

In each epoch, the goal is to agree on a set of (the indices of) at least $N - f$ dispersed blocks which are available for later retrieval. An epoch contains N instances of VID and BA. Let VID_i^e be the i-th ($1 \leq i \leq N$) VID instance of epoch e. VID_i^e is reserved for the i-th node to disperse (propose) its block.[3] Let BA_i^e be the i-th ($1 \leq i \leq N$) BA instance of epoch e. BA_i^e is for agreeing on whether to commit the block dispersed by the i-th node.

---

[3] Correct nodes ignore attempts from another node j ($j \neq i$) to disperse into VID_i^e by dropping $\texttt{Chunk}$ messages for VID_i^e from node j ($j \neq i$). Therefore, a Byzantine node cannot impersonate and disperse blocks on behalf of others.

Fig. 6 describes the single epoch protocol for the i-th node at epoch e. It begins by taking the block B_i^e to be proposed for this epoch, and dispersing it for epoch e through VID_i^e. Note that every block in the system is dispersed using a unique VID instance identified by its epoch number and proposing node.

Nodes now need to decide which blocks get committed in this epoch, and they should only commit blocks that have been successfully dispersed. Because there are potentially f Byzantine nodes, we cannot wait for all N instances of VID to complete because Byzantine nodes may never initiate their VID `Disperse`. On the other hand, nodes cannot simply wait for and commit the first $N - f$ VIDs to `Complete`, because VID instances may `Complete` in different orders at different nodes (hence correct nodes would not be guaranteed to commit the same set of blocks). DispersedLedger uses a strategy first proposed in [4]. Nodes use BA_i^e to explicitly agree on whether to commit B_i^e (which should be dispersed in VID_i^e). Correct nodes input 1 into BA_i^e only when VID_i^e `Completes`, so BA_i^e outputs 1 only if B_i^e *is* available for later retrieval. When $N - f$ BA instances have output 1, nodes give up on waiting for any more VID to `Complete`, and input 0 into the remaining BAs to explicitly signal the end of this epoch. This is guaranteed to happen because VID instances of the $N - f$ correct nodes will always `Complete` by the Termination property (§3.1). Once the set of committed blocks are determined, nodes can start retrieving the full blocks. After all blocks have been downloaded, a node sorts them by index number and delivers (executes) them in order.

The single-epoch DispersedLedger protocol is readily chained together epoch by epoch to achieve full SMR, as pictured in Fig. 5. At the beginning of every epoch, a node takes transactions from the head of the input buffer to form a block. After every epoch, a node checks if its block is committed. If not, it puts the transactions in the block back to the input buffer and proposes them in the next epoch. Also, a node delivers epoch e only after it has delivered all previous epochs.

4.3 Inter-node Linking

Motivation. An important limitation of the aforementioned single-epoch protocol (and all protocols with a similar construction [15, 31]) is that not all proposed blocks from correct nodes are committed in an epoch. An epoch only guarantees to commit $N - f$ proposed blocks, out of which $N - 2f$ are guaranteed to come from correct nodes. In other words, at most f blocks proposed by correct nodes are dropped every epoch. Dropped blocks can happen *with or without* adversarial behavior. Transmitting such blocks wastes bandwidth, for example, reducing HoneyBadger's throughput by 25% in our experiments (§6.2). To make the matter worse, the adversary (if present) can determine which blocks to drop [31], i.e. at most f correct servers can be *censored* such that *no* block from these servers gets committed. HoneyBadger provides a partial mitigation by keeping the proposed blocks encrypted until they are committed so that the adversary cannot censor
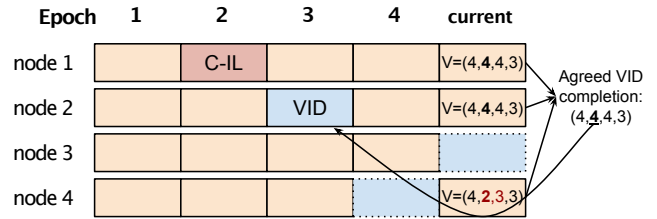
Figure 7: An example of commits by inter-node linking where $N = 4, f = 1$. Each box indicates a block proposed by a node at an epoch. Orange blocks are committed by BA. "VID" indicates that the block is dispersed but not committed. "C-IL" indicates a block committed by inter-node linking. Blue dotted boxes indicate a VID in progress. In the current epoch, after delivering the blocks from node 1, 2, and 4, the block proposed in epoch 3 by node 2 will be delivered by inter-node linking.

blocks by their content. The adversary *can*, however, censor blocks based on the proposing node.[4] This is unacceptable for consortium blockchains (§2.4), because the adversary could censor *all* transactions from certain (up to f) organizations. Moreover, HoneyBadger's mitigation relies on threshold cryptography, which incurs a high computational cost [15].

Our solution. We propose a novel solution to this problem, called *inter-node linking*, that guarantees *all* blocks from correct nodes are committed. Inter-node linking eliminates any censorship or bandwidth waste, and is readily applicable to similarly constructed protocols like HoneyBadger and BEAT. Notice that a block not committed by BA in a given epoch may still finish its VID. For example, in Fig. 7, the block proposed by node 2 in epoch 3 was dispersed but did not get selected by BA in that epoch. The core idea is to have nodes identify such blocks and deliver them in a consistent manner in later epochs.

Each node i keeps track of which VID instances have `Completed`, in the form of an array V_i^e of size N, which stores the local view at that node. When node i starts epoch e, it populates $V_i^e[j]$ (for all $1 \leq j \leq N$) with the largest epoch number such that all node j's VID instances up to epoch $V_i^e[j]$ have completed. For example, in Fig. 7, $(4, 4, 4, 3)$ would be a valid array V for the current epoch, and would indicate that node 2's VID for epoch 3 has completed but node 4's VID in epoch 4 has not.

Each node i reports its local array V_i^e in the block B_i^e it proposes in each epoch (in addition to the normal block content). As shown in Fig. 7, the BA mechanism then commits at least $N - f$ blocks in each epoch. During retrieval for epoch e, a node first retrieves the blocks committed by BA in epoch e and delivers (executes) them as in the single-epoch protocol (§4.2). It then extracts the set of V arrays in the committed blocks, i.e. $\{V_j^e \mid j \in S_i^e\}$, and combines the information across these arrays to determine additional blocks that it should retrieve (and deliver) in this epoch. Note that $S_i^e = S_j^e$ for any two correct nodes i, j due to the Agreement property of BA, so all correct nodes

---

[4]HoneyBadger suggests sending transactions to all nodes to prevent censorship, but this isn't possible for consortium blockchains and still wastes bandwidth due to dropped blocks (§6.2).

will use the same set of observations and get the same result.[5]

Using the committed V arrays, the inter-node linking protocol computes an epoch number $E^e[j]$ for each node j. This is computed locally by each node i, but we omit the index i since all (correct) nodes compute the same value. Each node then retrieves and delivers (executes) all blocks from node j until epoch $E^e[j]$. To ensure total order, nodes sort the blocks, first by epoch number then by node index. They also keep track of blocks that have been delivered so that no block is delivered twice.

In computing $E^e[j]$, we must be careful to not get misled by Byzantine nodes who may report arbitrary data in their V arrays. For example, naively taking the largest value reported for node j across all V arrays, i.e., $\max_{k \in S_i^e} V_k^e[j]$, would allow a Byzantine node to fool others into attempting to retrieve blocks that do not exist. Instead, we take the $(f+1)^{\text{th}}$-largest value; this guarantees that at least one correct node i has reported in its array V_i^e that node j has completed all its VIDs up to epoch $E^e[j]$. Recall that by the Availability property of VID (§3.1), this ensures that these blocks are available for retrieval. Also, since all correct blocks eventually finish VID (Termination property), all of them will eventually be included in E^e and get delivered. We provide pseudocode for the full DispersedLedger protocol in Appendix C.

4.4 Correctness of DispersedLedger

We now analyze the correctness of the DispersedLedger protocol by showing it guarantees the three properties required for BFT (§2.1). Full proof is in Appendix D.

Agreement and Total Order (Theorem D.7). Transactions are embedded in blocks, so we only need to show Agreement and Total Order of *block* delivery at each correct node. Blocks may get committed and delivered through two mechanisms: BA and inter-node linking. First consider blocks committed by BA. BA's Agreement and VID's Correctness properties guarantee that (i) all correct nodes will retrieve the same set of blocks for each epoch, and (ii) they will download the same content for each block. Now consider the additional blocks committed by inter-node linking. As discussed in §4.3, correct nodes determine these blocks based on identical information (V arrays) included in the blocks delivered by BA. Hence they all retrieve and deliver the same set of blocks (Lemma D.2). Also, all correct nodes use the same sorting criteria (BA-delivered blocks sorted by node index, followed by inter-node-linked blocks sorted by epoch number and node index), so they deliver blocks in the same order.

Validity (Theorems D.5, D.6). Define "correct transactions" as ones submitted by correct clients to correct nodes (servers). We want to prove every correct transaction is eventually delivered (executed). This involves two parts: (i) correct nodes do not hang, so that every correct transaction eventually gets proposed in some correct block (Theorem D.5); (ii) all

correct blocks eventually get delivered (Theorem D.6).

For part (i), note that all BAs eventually Output, since in every epoch at least $N-f$ BAs will Output(1) (Lemma D.3), and then all correct nodes will Input(0) to the remaining BAs and drive them to termination. Further, all blocks selected by BA or inter-node linking are guaranteed to be successfully dispersed, so Retrieve for them will eventually finish. By BA's Validity property, a BA only produces Output(1) when some correct node has Input(1), which can only happen if that node sees the corresponding VID Complete. Also, as explained in §4.3, inter-node linking only selects blocks that at least one correct node observes to have finished dispersal (Lemma D.4). By the Availability property of VID (§3.1), all these blocks are available for retrieval. For part (ii), note that all correct blocks eventually finish VID (Termination property). The inter-node linking protocol will therefore eventually identify all such blocks to have completed dispersal (Lemma D.4) and deliver them (if not already delivered by BA).

4.5 Practical Considerations

Running multiple epochs in parallel. In DispersedLedger, nodes perform dispersal sequentially, proceeding to the dispersal phase for the next epoch as soon as the dispersal for the current epoch has completed (all BA instances have Output). On the other hand, the retrieval phase of each epoch runs asynchronously at all nodes. To prevent slow nodes from stalling the progress of fast nodes, it is important that they participate in dispersal at as high a rate as possible, using only remaining bandwidth for retrieval. This effectively requires prioritizing dispersal traffic over retrieval traffic when there is a network bottleneck. Furthermore, a node can retrieve blocks from multiple epochs in parallel (e.g., to increase network utilization), but it must always deliver (execute) blocks in a serial order. Ideally, we want to fully utilize the network but prioritize traffic for earlier epochs over later epochs to minimize delivery latency. Mechanisms to enforce prioritization among different types of messages are implementation-specific (§5).

Constantly-slow nodes. Since DispersedLedger decouples the progress of fast and slow nodes, a natural question is: what if some nodes are constantly slow and do not have a chance to catch up? The possibility of some nodes constantly lagging behind is a common concern for BFT protocols. A BFT protocol cannot afford to wait for the slowest servers, because they could be Byzantine servers trying to stall the system [20]. Therefore the slow servers (specifically the f slowest servers) can be left behind, unable to catch up. Essentially, there is a tension between accommodating servers that are correct but slow, and preventing Byzantine nodes from influencing the system.

DispersedLedger expands this issue beyond the f slowest servers. We discuss two simple mitigations. First, the system designer could mandate a minimum average bandwidth per node such that all correct nodes can support the target system throughput over a certain timescale T. Every node

---

[5]If a particular Retrieve returns string "BAD_UPLOADER" or the block is ill formatted, we use array $[\infty, \infty, ..., \infty]$ as the observation.

must support the required bandwidth over time T but can experience lower bandwidth temporarily without stalling other nodes. Second, correct nodes could simply stop proposing blocks when too far behind, e.g., if their retrieval is more than P epochs behind the current epoch ($P = 1$ is the same as HoneyBadger). If enough nodes fall behind and stop proposing, it automatically slows down the system. A designer can choose parameters T or P to navigate the tradeoff between bandwidth variations impacting system throughput and how far behind nodes can get.

Spam transactions. In DispersedLedger, nodes do not check the validity of blocks they propose, deferring this check to the retrieval phase. This creates the possibility of malicious servers or clients spamming the system with invalid blocks.

Server-sent spam cannot be filtered even in conventional BFT protocols, because by the time other servers download the spam blocks, they have already wasted bandwidth. Similarly, HoneyBadger must perform BA (and incur its compute and bandwidth cost) regardless of the validity of the block, because by design, all BAs must eventually finish for the protocol to make progress [31]. Therefore, server-sent spam harms DispersedLedger and HoneyBadger equally. Fortunately, server-sent spam is bounded by the fraction of Byzantine servers (f/N).

On the other hand, client-sent spam is not a major concern in consortium blockchains (§2.1). In consortium blockchains, the organization is responsible for its clients, and a non-Byzantine organization would not spam the system.[6] For these reasons, some BFT protocols targeting consortium blockchains such as HyperLedger Fabric [2] forgo transaction filtering prior to broadcast for efficiency and privacy gains.

In more open settings, where clients are free to contact any server, spamming is a concern. A simple modification to the DispersedLedger protocol enables the same level of spam filtering as HoneyBadger. Correct nodes simply stop proposing new transactions when they are lagging behind in retrieval. Instead, they propose an empty block (with no transactions) to participate in the current epoch. In this way, correct nodes only propose transactions when they can verify them. Empty blocks still incur some overhead, so a natural question is: what is the performance impact of these empty blocks? Our results show that it is minor and this variant of DispersedLedger, which we call "DL-Coupled", retains most of the performance benefits (§6.2).

5 Implementation

We implement DispersedLedger in 8,000 lines of Go. The core protocol of DispersedLedger is modelled as 4 nested IO automata: `BA`, `VID`, `DLEpoch`, and `DL`. `BA` implements the binary agreement protocol proposed in [32]. `VID` implements our verifiable information dispersal protocol AVID-M

described in §3.3. We use a pure-Go implementation of Reed-Solomon code [36] for encoding and decoding blocks, and an embedded key-value storage library [23] for storing blocks and chunks. `DLEpoch` nests N instances of `VID` and `BA` to implement one epoch of DispersedLedger (§4.2). Finally, `DL` nests multiple instances of `DLEpoch` and the inter-node linking logic (§4.3) to implement the full protocol.

Traffic prioritization. Prioritizing dispersal traffic over retrieval is made complicated because nodes cannot be certain of the bottleneck capacity for different messages and whether they share a common bottleneck. For example, rate-limiting the low-priority traffic may result in under-utilization of the network. Similarly, simply enforcing prioritization between each individual pair of nodes may lead to significant priority inversion if two pairs of nodes share the same bottleneck. In our implementation, we use a simple yet effective approach to achieve prioritization in a work conserving manner (without static rate limits) inspired by MulTcp [13]. For each pair of nodes, we establish two connections, and we modify the parameters of the congestion control algorithm of one connection so that it behaves like T ($T > 1$) connections . We then send high-priority traffic on this connection, and low-priority traffic on the other (unmodified) connection. At all bottlenecks, the less aggressive low-priority connection will back off more often and yield to the more aggressive high-priority connection. On average, a high-priority connection receives T times more bandwidth than a competing low-priority connection at the same bottleneck.[7] Note that in DispersedLedger, high-priority traffic consists of only a tiny fraction of the total traffic that a node handles ($1/20$ to $1/10$ in most cases as shown in §6.4), and its absolute bandwidth is low. Therefore our approach will not cause congestion to other applications competing at the same bottleneck. In our system, we set $T = 30$. We use QUIC as the underlying transport protocol and modify the `quic-go` [12] library to add the knob T for tuning the congestion control.

To prioritize retrieval traffic by epoch, we order retrieval traffic on a per-connection basis by using separate QUIC streams for different epochs. We modify the scheduler `quic-go` [12] to always send the stream with the lowest epoch number.

Rate control for block proposal. DispersedLedger requires some degree of batching to amortize the fixed cost of BA and VID. However, if unthrottled, nodes may propose blocks too often and the resulting blocks could be very small, causing low bandwidth efficiency. More importantly, since dispersal traffic is given high priority, the system may use up all the bandwidth proposing inefficient small blocks and leave no bandwidth for block retrieval. To solve this problem, our implementation employs a simple form of adaptive batching [29]. Specifically, we limit the block proposal rate using Nagle's algorithm [33]. A node only proposes a new block if

---

[6] A Byzantine organization could of course spam, but this is the same as the server-sent spamming scenario, in which DispersedLedger is no worse than HoneyBadger.

[7] Similar approaches have been used in other usecases to control bandwidth sharing among competing flows [34].

(i) a certain duration has passed since the last block was proposed, or (ii) a certain amount of data has accumulated to be proposed in the next block. In our implementation, we use 100 ms as the delay threshold, and 150 KB as the size threshold. This setup works well for all of our evaluation experiments.

6 Evaluation

Our evaluation answers the following questions:

1. What is the throughput and the latency of DispersedLedger in a realistic deployment?
2. Is DispersedLedger able to consistently achieve good throughput regardless of network variations?
3. How does the system scale to more nodes?

We compare DispersedLedger (DL) with the original HoneyBadger (HB) and our optimized version: HoneyBadger-Link. HoneyBadger-Link (HB-Link) combines the inter-node linking in DispersedLedger with HoneyBadger, so that every epoch, all (instead of $N-2f$) honest blocks are guaranteed to get into the ledger. We also experiment with DL-Coupled, a variant of DispersedLedger where nodes only propose new transactions when they are up-to-date with retrievals (§4.5).

6.1 Experimental Setup

We run our evaluation on AWS EC2. In our experiments, every node is hosted by an EC2 `c5d.4xlarge` instance with 16 CPU cores, 16 GB of RAM, 400 GB of NVMe SSD, and a 10 Gbps NIC. The nodes form a fully connected graph, i.e. there is a link between every pair of nodes. We run our experiments on two different scenarios. First, a *geo-distributed* scenario, where we launch VMs at 16 major cities across the globe, one at each city. We don't throttle the network. This scenario resembles the typical deployment of a consortium blockchain. In addition, we measure the throughput of the system on another testbed on Vultr (details are in Appendix A.2). Second, a *controlled* scenario, where we start VMs in one datacenter and apply artificial delay and bandwidth throttling at each node using Mahimahi [35]. Specifically, we add a one-way propagation delay of 100 ms between each pair of nodes to mimic the typical latency between distant major cities [38], and model the ingress and egress bandwidth variation of each node as independent Gauss-Markov processes (more details in §6.3). This controlled setup allows us to precisely define the variation of the network condition and enables fair, reproducible evaluations. Finally, to generate the workload for the system, we start a thread on each node that generates transactions in a Poisson arrival process.

6.2 Performance over the Internet

First, we measure the performance of DispersedLedger on our geo-distributed testbed and compare it with HoneyBadger.
Throughput. To measure the throughput, we generate a high load on each node to create an infinitely-backlogged system, and report the rate of confirmed transactions at each node. Because the internet bandwidth varies at different locations, we expect the measured throughput to vary as well. Fig. 8
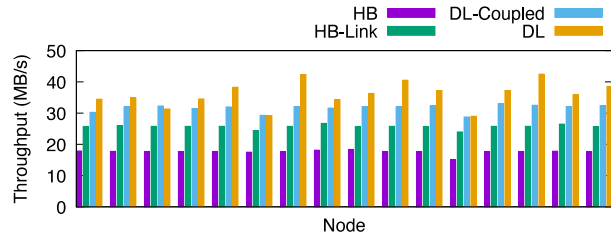
Figure 8: Throughput of each server running different protocols on the geo-distributed setting.
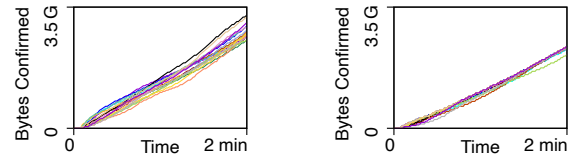
(a) DispersedLedger   **(b)** HoneyBadger with linking

Figure 9: The amount of confirmed data over time when running DispersedLedger and HoneyBadger with inter-node linking on the geo-distributed testbed, plotted on the same scale. Each line represents one server.

shows the results. DispersedLedger achieves on average 105% better throughput than HoneyBadger. To confirm that our scheme is robust, we also run the experiment on another testbed using a low-cost cloud vendor. Results in §A.2 show that DispersedLedger significantly improves the throughput in that setting as well.

DispersedLedger gets its throughput improvement mainly for two reasons. First, inter-node linking ensures all blocks that successfully finish VID get included in the ledger, so no bandwidth is wasted. In comparison, in every epoch of HoneyBadger at most f blocks may *not* get included in the final ledger. The bandwidth used to broadcast them is therefore wasted. As a result, inter-node linking provides at most a factor of $N/(N-f)$ improvement in effective throughput. To measure the gain in the real-world setting, we modify HoneyBadger to include the same inter-node linking technique and measure its throughput. Results in Fig. 8 show that enabling inter-node linking provides a 45% improvement in throughput on our geo-distributed testbed.

Second, confirmation throughput at different nodes are decoupled, so temporary slowdown at one site will not affect the whole system. Because the system is deployed across the WAN, there are many factors that could cause the confirmation throughput of a node to fluctuate: varying capacity at the network bottleneck, latency jitters, or even behavior of the congestion control algorithm. In HoneyBadger, the confirmation progress of all but the f slowest nodes are coupled, so at any time the whole system is only as fast as the $f+1$-slowest node. DispersedLedger does not have this limitation. Fig. 9 shows an example: DispersedLedger allows each node to always run at its own capacity. HoneyBadger couples the performance of most servers together, so all servers can only progress at the same, limited rate. In fact, notice that

every node makes more progress with DispersedLedger compared to HoneyBadger (with linking) over the 2 minutes shown. The reason is that with HoneyBadger, different nodes become the straggler (the $(f+1)^{\text{th}}$-slowest node) at different time, stalling all other nodes. But with DispersedLedger, a slow node whose bandwidth improves can accelerate and make progress independently of others, making full use of time periods when it has high bandwidth. Fig. 8 shows that DispersedLedger achieves 41% better throughput compared to HoneyBadger with linking due to this improvement.

Finally, DL-Coupled is 12% slower than DL on average, but it still achieves 80% and 23% higher throughput on average than HoneyBadger and HoneyBadger with linking. Recall that DL-Coupled constrains nodes that can propose new transactions to prevent spamming attacks. The result shows that in open environments where spamming is a concern, DL-Coupled can still provide significant performance gains. In the rest of the evaluation, we focus on DL (without spam mitigation) to investigate our idea in its purest form.

Latency. Confirmation latency is defined as the elapsed time from a transaction entering the system to it being delivered. Similar to the throughput, the confirmation latency at different servers varies due to heterogeneity of the network condition. Further, for a particular node, we only calculate the latency of the transactions that this node *itself* generates, i.e. *local* transactions. This is a somewhat artificial metric, but it helps isolate the latency of each server in HoneyBadger and makes the results easier to understand. In HoneyBadger, a slow node only proposes a new epoch after it has confirmed the previous epoch, so the rate it proposes is coupled with the rate it confirms, i.e. it proposes 1 block after downloading $O(N)$ blocks. Due to this reason, an overloaded node does not have the capacity to even *propose* all the transactions it generates, and whatever transaction it proposes will be stale. When these stale transactions get confirmed at a fast node, the latency (especially the tail latency) at the fast nodes will suffer. Note that DispersedLedger does not have this problem, because all nodes, even overloaded ones, propose new transactions at a rate limited only by the egress bandwidth. In summary, choosing this metric is only advantageous to HoneyBadger, so the experiment remains fair. In Appendix §A.1, we provide further details and report the latency of all servers calculated for both local only, and all transactions.

We run the system at different loads and report the latency at each node. In Fig. 10, we focus on two datacenters: Mumbai, which has limited internet connection, and Ohio, which has good internet connection. We first look at the median latency. At low load, both HoneyBadger and DispersedLedger have similarly low median latency. But as we increase the load from 6 MB/s to 15 MB/s, the median latency of HoneyBadger increases almost linearly from around 800 ms to 3000 ms. This is because in HoneyBadger, proposing and confirming an epoch are done in lockstep. As the load increases, the proposed block becomes larger and
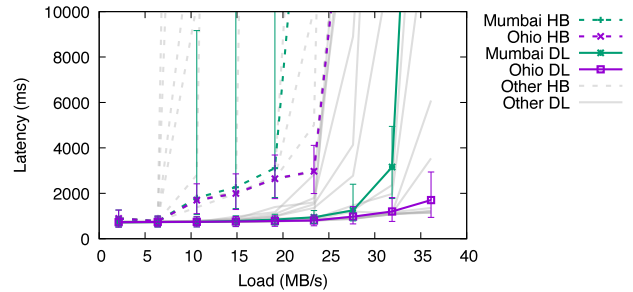
Figure 10: The median latency of DispersedLedger (solid) and HoneyBadger (dash) under different offered load. Error bar shows the 5-th and the 95-th percentiles. Two locations with good (Ohio) and limited (Mumbai) internet connection are highlighted.

takes longer to confirm. This in turn causes more transactions to be queued for the next block so the next proposed block remains large. Actually, the batch (all blocks in an epoch) size of HoneyBadger increases from 3.4 MB to 42.5 MB (200 KB to 2.5 MB per block) as we increase the load from 6 MB/s to 15 MB/s. Note that the block size is not chosen by us, but is naturally found by the system itself. In comparison, the latency of DispersedLedger only increases by a bit when the load increases, from 730 ms to 830 ms as we increase the load from 2 MB/s to 23 MB/s. The batch size ranges between 0.85 MB to 11.9 MB (50 KB to 700 KB per block).

We now look at the tail latency, which is important for service quality. At low load (6 MB/s), the 99-th percentile latency of DispersedLedger is 1000 ms across all servers, while that of HoneyBadger ranges from 1500 ms to 4500 ms. It suggests that DispersedLedger is more stable. As we increase the load, the tail (95-th percentile) latency of the Mumbai server immediately goes up. This is because HoneyBadger does not guarantee all honest blocks to be included in the ledger, and slow nodes are more likely to see their blocks being dropped from an epoch. When it happens, the node has to re-propose the same block in the next epoch, causing significant delay to the block. We note that the tail latency of the Ohio server goes up as well. In comparison, the tail latency of DispersedLedger at both Mumbai and Ohio stays low until very high load.

6.3 Controlled experiments

In this experiment, we run a series of tests in the controlled setting to verify if DispersedLedger achieves its design goal: achieving good throughput regardless of network variation. We start 16 servers in one datacenter, and add an artificial one-way propagation delay of 100 ms between each pair of servers to emulate the WAN latency. We then generate synthetic traces for each server that independently caps the ingress and egress bandwidth of the server. For each set of traces, we measure the throughput of DispersedLedger and HoneyBadger.

Spatial variation. This is the situation where the bandwidth varies across different nodes but stays the same over time. For the *i*-th node ($0 \leq i < 16$), we set its bandwidth to constantly be $10 + 0.5i$ MB/s. Fig. 11a shows that the throughput of
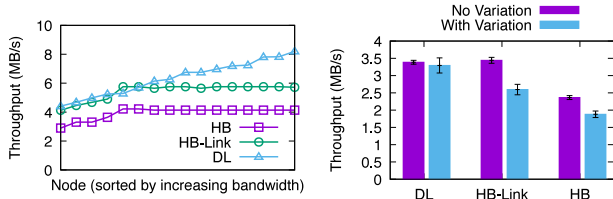
(a) Spatial variation   **(b)** Temporal variation

Figure 11: Throughput of HoneyBadger (HB), HoneyBadger with linking (HB-Link), and DispersedLedger (DL) in the controlled experiments. Error bars in (b) show the standard deviation.

Figure 12: Throughput at different cluster size and block size. Error bars show the standard deviation.

Figure 13: Fraction of dispersal traffic versus total traffic at different scale and block size.

HoneyBadger (with or without linking) is capped at the bandwidth of the fifth slowest server, and the bandwidth available at all faster servers are not utilized. In comparison, the throughput of DispersedLedger at different servers are fully decoupled. The achieved bandwidth is proportional to the available bandwidth at each server. DispersedLedger achieves this because it decouples block retrieval at different servers.

Temporal variation. We now look at the scenario where the bandwidth varies over time, and show that DispersedLedger is robust to network fluctuation. We model the bandwidth variation of each node as independent Gauss-Markov processes with mean b, variance σ, and correlation between consecutive samples α, and generate synthetic traces for each node by sampling from the process every 1 second. Specifically, we set $b = 10$ MB/s, $\sigma = 5$ MB/s, $\alpha = 0.98$ and generate a trace for each server, i.e. the bandwidth of each server varies independently but have the same distribution with mean bandwidth 10 MB/s. (We show an example of such trace in §A.3.) As a comparison, we also run an experiment when the bandwidth at each server does not fluctuate and stays at 10 MB/s. In our implementation (for all protocols), a node notifies others when it has decoded a block to stop sending more chunks. This optimization is less effective when all nodes have exactly the same fixed bandwidth because all chunks for a block will arrive at roughly the same time. So in this particular experiment, we disable this optimization to enable an apple-to-apple comparison of the fixed and variable bandwidth scenarios. Fig. 11b shows that as we introduce temporal variation of the network bandwidth, the throughput of DispersedLedger stays the same. This confirms that DispersedLedger is robust to network fluctuation. Meanwhile, the throughput of HoneyBadger and HoneyBadger with linking dropped by 20% and 25% respectively.

6.4 Scalability

In this experiment, we evaluate how DispersedLedger scales to a large number of servers. As with many evaluations of BFT protocols [31, 39], we use cluster sizes ranging from 16 to 128.

Throughput. We first measure the system throughput at different cluster size N. For this experiment, we start all the servers in the same datacenter with a 100 ms one-way propagation delay on each link and a 10 MB/s bandwidth cap on each server. We also fix the block size to 500 KB
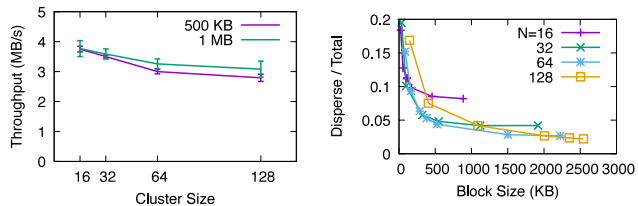
and 1 MB. Fig. 12 shows that the system throughput slightly drops when N grows 8 times bigger from 16 nodes to 128 nodes. This is because the BA in the dispersal phase has a per-node cost of $O(N^2)$. With a constant block size, the messaging overhead takes a larger fraction as N increases. We notice that increasing the block size helps amortize the cost of VID and BA, and results in better system throughput. **Traffic for block dispersal.** A metric core to the design of DispersedLedger is the amount of data a node has to download in order to participate in block dispersal, i.e. dispersal traffic. More precisely, we are interested in the *ratio* of dispersal traffic to the total traffic (dispersal plus retrieval). The lower this ratio, the easier it is for slow nodes to keep up with block dispersal, and the better DispersedLedger achieves its design goal. Fig. 13 shows this ratio at different scales and block sizes. First, we observe that increasing the block size brings down the fraction of dispersal traffic. This is because a large block size amortizes the fixed cost in VID and BA. Meanwhile, increasing the cluster size reduces the lower bound on the fraction of dispersal traffic. This is because in the VID phase, every node is responsible for an $1/(N-2f)$ slice of each block, and increasing N brings down this fraction.

7 Conclusion

We presented DispersedLedger, a new asynchronous BFT protocol that provides near-optimal throughput under fluctuating network bandwidth. DispersedLedger is based on a novel restructuring of BFT protocols that decouples agreement from the bandwidth-intensive task of downloading blocks. We implement a full system prototype and evaluate DispersedLedger on two testbeds across the real internet and a controlled setting with emulated network conditions. Our results on a wide-area deployment across 16 major cities show that DispersedLedger achieves $2\times$ better throughput and 74% lower latency compared to HoneyBadger. Our approach could be applicable to other BFT protocols, and enables new applications where resilience to poor network condition is vital.

Acknowledgments

References

[1] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and M. Yin. Sync HotStuff: Simple and practical synchronous state machine replication. In *2020 IEEE Symposium on Security and Privacy*, pages 106–118. IEEE, 2020.

[2] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick. Hyperledger Fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, pages 30:1–30:15. ACM, 2018.

[3] M. Belotti, N. Božić, G. Pujolle, and S. Secci. A vademecum on blockchain technologies: When, which, and how. *IEEE Communications Surveys & Tutorials*, 21(4):3796–3838, 2019.

[4] M. Ben-Or, B. Kelmer, and T. Rabin. Asynchronous secure computations with optimal resilience. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 183–192. ACM, 1994.

[5] E. Buchman. *Tendermint: Byzantine fault tolerance in the age of blockchains*. PhD thesis, University of Guelph, 2016.

[6] V. Buterin. On public and private blockchains. https://blog.ethereum.org/2015/08/07/on-public-and-private-blockchains/. Accessed: 2021-08-15.

[7] V. Buterin and V. Griffith. Casper the friendly finality gadget. *arXiv:1710.09437v4*, 2019.

[8] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup. Secure and efficient asynchronous broadcast protocols. In *Advances in Cryptology — CRYPTO 2001*, pages 524–541. Springer, 2001.

[9] C. Cachin and J. A. Poritz. Secure intrusion-tolerant replication on the internet. In *2002 International Conference on Dependable Systems and Networks*, pages 167–176. IEEE, 2002.

[10] C. Cachin and S. Tessaro. Asynchronous verifiable information dispersal. In *24th IEEE Symposium on Reliable Distributed Systems*, pages 191–201. IEEE, 2005.

[11] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pages 173–186. USENIX Association, 1999.

[12] L. Clemente. quic-go: A QUIC implementation in pure Go. https://github.com/lucas-clemente/quic-go.

[13] J. Crowcroft and P. Oechslin. Differentiated end-to-end internet services using a weighted proportional fair sharing TCP. *ACM SIGCOMM Computer Communication Review*, 28(3):53–69, 1998.

[14] M. Du, Q. Chen, J. Xiao, H. Yang, and X. Ma. Supply chain finance innovation using blockchain. *IEEE Transactions on Engineering Management*, 67(4):1045–1058, 2020.

[15] S. Duan, M. K. Reiter, and H. Zhang. BEAT: Asynchronous BFT made practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2028–2041. ACM, 2018.

[16] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[17] A. Gągol and M. Świętek. Aleph: A leaderless, asynchronous, Byzantine fault tolerant consensus protocol. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, pages 214–228. ACM, 2019.

[18] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich. Algorand: Scaling Byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 51–68. ACM, 2017.

[19] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. Efficient Byzantine-tolerant erasure-coded storage. In *2004 International Conference on Dependable Systems and Networks*, pages 135–144. IEEE, 2004.

[20] G. G. Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. Reiter, D.-A. Seredinschi, O. Tamir, and A. Tomescu. SBFT: a scalable and decentralized trust infrastructure. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 568–580. IEEE, 2019.

[21] J. Hendricks, G. R. Ganger, and M. K. Reiter. Verifying distributed erasure-coded data. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 139–146. ACM, 2007.

[22] N. Kabra, P. Bhattacharya, S. Tanwar, and S. Tyagi. Mudrachain: Blockchain-based framework for automated cheque clearance in financial institutions. *Future Generation Computer Systems*, 102:574–587, 2020.

[23] A. Krylysov. pogreb: Embedded key-value store for read-heavy workloads written in Go. https://github.com/akrylysov/pogreb.

[24] K. Kursawe and V. Shoup. Optimistic asynchronous atomic broadcast. In *Automata, Languages and Programming*, pages 204–215. Springer, 2005.

[25] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. In *Concurrency: The Works of Leslie Lamport*, pages 203–226. ACM, 2019.

[26] J. Liu, T. Liang, R. Sun, X. Du, and M. Guizani. A privacy-preserving medical data sharing scheme based on consortium blockchain. In *2020 IEEE Global Communications Conference*, pages 1–6. IEEE, 2020.

[27] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 17–30. ACM, 2016.

[28] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[29] J. C. McCullough, J. Dunagan, A. Wolman, and A. C. Snoeren. Stout: An adaptive interface to scalable cloud storage. In *2010 USENIX Annual Technical Conference*. USENIX Association, 2010.

[30] R. C. Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptology — CRYPTO '87*, pages 369–378. Springer, 1987.

[31] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song. The honey badger of BFT protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 31–42. ACM, 2016.

[32] A. Mostéfaoui, M. Hamouma, and M. Raynal. Signature-free asynchronous Byzantine consensus with $t < n/3$ and $O(n^2)$ messages. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, pages 2–9. ACM, 2014.

[33] J. Nagle. Congestion control in IP/TCP internetworks. RFC 896, RFC Editor, 1984.

[34] V. Nathan, V. Sivaraman, R. Addanki, M. Khani, P. Goyal, and M. Alizadeh. End-to-end transport for video QoE fairness. In *Proceedings of the 2019 Conference of the ACM Special Interest Group on Data Communication*, pages 408–423. ACM, 2019.

[35] R. Netravali, A. Sivaraman, S. Das, A. Goyal, K. Winstein, J. Mickens, and H. Balakrishnan. Mahimahi: Accurate record-and-replay for HTTP. In *2015 USENIX Annual Technical Conference*, pages 417–429. USENIX Association, 2015.

[36] K. Post. reedsolomon: Reed-Solomon erasure coding in Go. https://github.com/klauspost/reedsolomon.

[37] M. O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2):335–348, 1989.

[38] WonderNetwork. Global ping statistics. https://wondernetwork.com/pings/. Accessed: 2021-08-15.

[39] M. Yin, D. Malkhi, M. K. Reiter, G. Golan-Gueta, and I. Abraham. HotStuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356. ACM, 2019.

[40] R. Zhang, R. Xue, and L. Liu. Security and privacy on blockchain. *ACM Computing Surveys*, 52(3), 2019.

A Supplements to the Evaluations

A.1 Latency metric

Here we justify counting only local transactions when calculating the confirmation latency. As mentioned in §6.2, we choose this metric to prevent overloaded servers from impacting the latency (especially the tail latency) of non-overloaded servers. Fig. 14 shows the latency of DispersedLedger and HoneyBadger under two metrics: counting all transactions, and counting only local transactions. Each system is running near its capacity. We observe that the latency (both the median and the tail) of DispersedLedger is the same under the two metrics, so choosing to count only local transactions in no way helps our protocol. For HoneyBadger, we observe that by counting all transactions, the median latency of the overloaded servers decreased. This is because the overloaded servers cannot get their local transactions into the ledger (so the local transactions have high latency), but can confirm *some* transactions from other non-overloaded servers. The median latency mostly represents these non-local transactions. Still, these servers are overloaded, and the latency numbers are meaningless because they will increase as system runs for longer. So the latency metric does not matter for the overloaded servers. Meanwhile, we observe that the tail latency of HoneyBadger on non-overloaded servers worsens a lot as we switch to counting all transactions. This is due to the transactions proposed by the overloaded nodes, and is the main reason that we choose to count only local transactions. In summary, counting only local transactions for latency calculation does not improve the latency of DispersedLedger, but helps improve the tail latency of non-overloaded servers in HoneyBadger, so choosing this metric is fair.

A.2 Throughput on another testbed over the internet

To further confirm that DispersedLedger improves the throughput of BFT protocols when running over the internet, we build another testbed on a low-cost cloud provider called Vultr. We use the $80/mo plan with 6 CPU cores, 16 GB of RAM, 320 GB of SSD, and an 1 Gbps NIC. At the moment of the experiment, Vultr has 15 locations across the globe, and
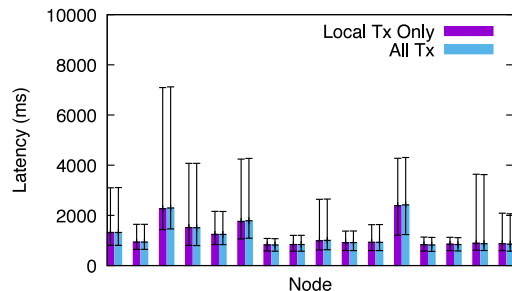
(a) DispersedLedger
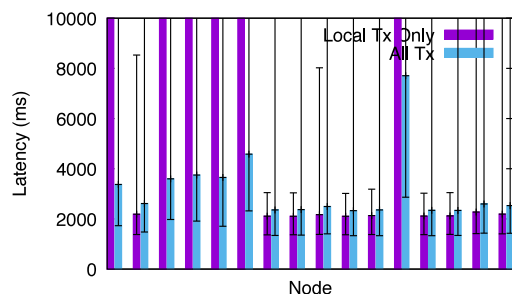
(b) HoneyBadger

Figure 14: Confirmation latency of DispersedLedger and HoneyBadger when counting all transactions (All Tx) or only local transactions. Each system runs near its capacity (14.8 MB/s for HoneyBadger and 23.4 MB/s for DispersedLedger). The error bar shows the 5-th and 95-th percentiles.
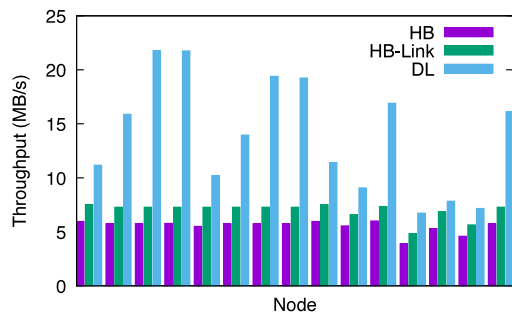
Figure 15: Throughput of each server running different protocols on the Vultr testbed. HB stands for HoneyBadger, HB-Link stands for HoneyBadger with inter-node linking, New stands for DispersedLedger.

we run one server at each location and perform the same experiment as in § 6.2. Fig. 15 shows the results. DispersedLedger improves the throughput by at least 50% over HoneyBadger.

A.3 Example trace of temporal variation

We provide in Fig. 16 an example of the synthetic bandwidth trace we used in the temporal variation scenario in §6.3.
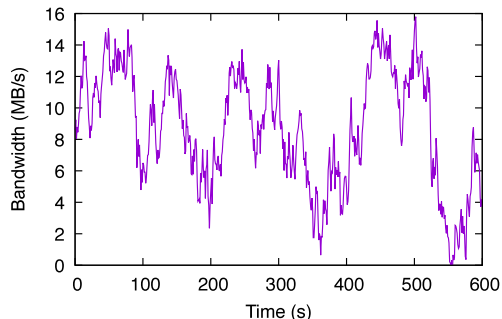
Figure 16: A bandwidth trace we used in the temporal variation scenario.

B Correctness proof of AVID-M

Notations. We use the symbol "·" as placeholders in message parameters to indicate "any". For example, $\text{Chunk}(r, \cdot, \cdot)$ means "Chunk messages with the first parameter set to r and the other two parameters set to any value".

Lemma B.1. *If a correct server sends* $\text{Ready}(r)$*, then at least one correct server has received* $N-f$ $\text{GotChunk}(r)$*.*

Proof. A correct server broadcasts $\text{Ready}(r)$ in two cases:
1. Having received $N-f$ $\text{GotChunk}(r)$ messages.
2. Having received $f+1$ $\text{Ready}(r)$ messages.

If a correct server sends out $\text{Ready}(r)$ for the aforementioned reason 1, then this already satisfies the lemma we want to prove. Now assume that a correct server sends $\text{Ready}(r)$ because it has received $f+1$ $\text{Ready}(r)$ (the aforementioned reason 2). Then there must exist a correct server which has sent out $\text{Ready}(r)$ because of the aforementioned reason 1. Otherwise, there can be at most f $\text{Ready}(r)$ messages (forged by the f Byzantine servers), and no correct server will ever send $\text{Ready}(r)$ because of reason 2, which contradicts with our assumption. So there exists a correct server that has received $N-f$ $\text{GotChunk}(r)$, and this satisfies the lemma. $\quad\square$

Theorem B.2 (Termination). *If a correct client invokes* Disperse *and no other client invokes* Disperse *on the same instance of VID, then all correct servers eventually* Complete *the dispersal.*

Proof. A correct client sends correctly encoded chunks to all servers. Let's assume the Merkle root of the chunks is r, then all correct servers eventually receive $\text{Chunk}(r, \cdot, \cdot)$. Because there is no other client invoking Disperse, it is impossible for a server to receive $\text{Chunk}(r', \cdot, \cdot)$ for any $r' \neq r$, and no correct server will ever broadcast $\text{GotChunk}(r')$ for any $r' \neq r$. So each correct server will send out $\text{GotChunk}(r)$. Eventually, all correct servers will receive $N-f$ $\text{GotChunk}(r)$.

All correct servers will broadcast $\text{Ready}(r)$ upon receiving these $N - f$ $\text{GotChunk}(r)$ messages or they have already sent $\text{Ready}(r)$. A correct server will Complete upon receiving $2f + 1$ $\text{Ready}(r)$. We have shown that all $N-f$

correct servers will eventually send Ready(r). Because $N - f \geq 2f + 1$, all correct servers will Complete. $\square$

Lemma B.3. *If a correct server has sent out* Ready(r)*, then no correct server will ever send out* Ready(r') *for any* $r' \neq r$.

Proof. Let's assume for contradiction that two messages Ready(r) and Ready(r') ($r \neq r'$) have both been sent by correct servers. By Lemma B.1, at least one correct server has received $N - f$ GotChunk(r), and at least one correct server has received $N - f$ GotChunk(r') ($r' \neq r$).

We obtain a contradiction by showing that the system cannot generate $N - f$ GotChunk(r) messages plus $N - f$ GotChunk(r') messages for the two correct servers to receive. Assume h GotChunk(r) messages come from correct servers, h' GotChunk(r') come from correct servers, and there are β Byzantine servers ($\beta \leq f$ by the definition of f). Then we have

$$h + \beta \geq N - f$$
$$h' + \beta \geq N - f.$$

A correct server do not broadcast *both* GotChunk(r) and GotChunk(r'), while a Byzantine server is free to send different GotChunk messages to different correct servers, so we have

$$h + h' + \beta \leq N.$$

These constraints imply

$$\beta \geq N - 2f.$$

However, $\beta \leq f$, so we must have $N \leq 3f$. This contradicts with our assumption of $N \geq 3f + 1$ in our security model (§2.4), so it is impossible, and the assumption must not hold. $\square$

Theorem B.4 (Agreement). *If some correct server* Complete*s the dispersal, then all correct servers will eventually* Complete *the dispersal.*

Proof. A correct server Completes if and only if it has received $2f + 1$ Ready(r) messages. We want to prove that in this situation, all correct servers will eventually *send* a Ready(r), so that they will all *receive* at least $2f + 1$ Ready(r) messages needed to Complete.

We now assume a correct server has Completed after receiving $2f + 1$ Ready(r). Out of these messages, at least $f + 1$ must be broadcast from correct servers, so all correct servers will eventually receive these Ready(r). A correct server will send out Ready(r) upon receiving $f + 1$ Ready(r), so all correct servers will do so upon receiving the aforementioend $f + 1$ Ready(r) messages.

Because all correct servers will send Ready(r), eventually all correct servers will receive $N - f$ Ready(r). Because $N - f \geq 2f + 1$, all of them will Complete. $\square$

Lemma B.5. *If a correct server has* Complete*d, then all correct servers eventually set the variable* ChunkRoot *to the same value.*

Proof. A correct server uses ChunkRoot to store the root of the chunks of the dispersed block, so we are essentially proving that all correct servers agree on this root. Assume that a server Completes, then it must have received $2f + 1$ Ready(r) messages. We now prove that no correct server can ever receive $2f + 1$ Ready(r') messages for any $r' \neq r$. Because a correct server has received $2f + 1$ Ready(r), there must be $f + 1$ correct servers who have broadcast Ready(r). By Lemma B.3, no correct server will ever broadcast Ready(r') for any $r' \neq r$, so a correct server can receive at most f Ready(r') for any $r' \neq r$, which are forged by the f Byzantine servers.

By Theorem B.4, all correct servers eventually Complete, so they must eventually receive $2f + 1$ Ready(r), and will each set ChunkRoot $= r$. $\square$

Theorem B.6 (Availability). *If a correct server has* Complete*d, and a correct client invokes* Retrieve*, it eventually reconstructs some block* B'.

Proof. The Retrieve routine returns at a correct client as long as it can collect $N - 2f$ ReturnChunk(r, C_i, P_i) messages with the same root r and valid proofs P_i. A correct server sends ReturnChunk(MyRoot,MyChunk,MyProof) to a client as long as it has MyRoot, MyChunk, MyProof, and ChunkRoot set, and MyRoot $=$ ChunkRoot. Here, a server uses MyRoot to store the root of the chunk it has received, uses MyChunk to store the chunk, and uses MyProof to store the Merkle proof (Fig. 3). We now prove that if any correct server Completes, at least $N - 2f$ correct servers will eventually meet this condition and send ReturnChunk to the client.

Assume that a correct server has Completed the VID instance with ChunkRoot set to r. Then, by Lemmas B.4, B.5, all correct servers will eventually Complete and set ChunkRoot $= r$. Also, this server must have received $2f + 1$ Ready(r) messages, out of which at least $f + 1$ must come from correct servers. According to Lemma B.1, at least one correct server has received $N - f$ GotChunk(r). At least $N - 2f$ GotChunk(r) messages must come from correct servers, so they each must have MyChunk, MyProof set, and have set MyRoot $= r$.

We have proved that at least $N - 2f$ correct servers will send ReturnChunk(r, C_i, P_i) messages. For each message sent by the i-th server (which is correct), P_i must be a valid proof showing C_i is the i-th chunk under root r, because the server has validated this proof. So the client will eventually obtain the $N - 2f$ chunks needed to reconstruct a block. $\square$

Lemma B.7. *Any two correct clients finishing* Retrieve *have their variable* ChunkRoot *set to the same value.*

Proof. A client uses variable ChunkRoot to store the root of the $N - 2f$ chunks it uses to reconstruct the block (Fig. 4),

so we are essentially proving that any two correct clients will use chunks under the same root when executing `Retrieve`. Let's assume for contradiction that two correct clients finish `Retrieve`, but have set `ChunkRoot` to r and r' respectively ($r \neq r'$). This implies that one client has received at least $N - 2f$ `ReturnChunk`$(r, \cdot, \cdot)$ messages, and the other has received $N - 2f$ `ReturnChunk`$(r', \cdot, \cdot)$ messages. Out of these messages, at least $N-3f$ `ReturnChunk`$(r,\cdot,\cdot)$ and at least $N-3f$ `ReturnChunk`$(r',\cdot,\cdot)$ are from correct servers (because $N \geq 3f + 1$ by our security assumptions in §2.4). Since a correct server ensures `MyRoot = ChunkRoot` and uses `MyRoot` as the first parameter of `ReturnChunk` messages, there must exist some correct server with `ChunkRoot` set to r, and some correct server with `ChunkRoot` set to r'. Also, since a correct server only sends `ReturnChunk` when it has `Completed`, there must be some server which has `Completed`. This contradicts with Lemma B.5, which states that all correct servers must have `ChunkRoot` set to the same value. The assumption must not hold. □

Extra notations. To introduce the following lemma, we need to define a few extra notations. Let `Encode`(B) be the encoding result of a block B in the form of an array of N chunks. Let `Decode`(C) be the decoding result (a block) of an array of $N-2f$ chunks. Let `MerkleRoot`(C) be the Merkle root of an array of chunks.

Lemma B.8. *For any array of N chunks C, exactly one of the following is true:*
1. *For any two subsets D_1, D_2 of $N - 2f$ chunks in C, `Decode`$(D_1) = $`Decode`$(D_2)$.*
2. *For any subset D of $N - 2f$ chunks in C, `MerkleRoot`$($`Encode`$($`Decode`$(D))) \neq $`MerkleRoot`$(C)$.*

Proof. We are proving that a set of chunks C is either:
1. Correctly encoded (consistent), so any subset of $N-2f$ chunks in C decode into the same block.
2. Or, no matter which subset of $N-2f$ chunks in C are used for decoding, a correct client can re-encode the decoded block, compute the Merkle root over the encoding result, and find it to be *different* from the Merkle root of C, and thus detect an encoding error.

Case 1: Consistent encoding. Assume for any subset D of $N-2f$ chunks in C, `Decode`$(D) = B$. We now want to prove that `MerkleRoot`$($`Encode`$($`Decode`$(D))) = $`MerkleRoot`$(C)$. By our assumption, `Encode`$($`Decode`$(D)) = $`Encode`$(B)$, so we only need to show $C = $`Encode`$(B)$. This is clearly true by the definition of erasure code: the `Encode` function encodes B into a set of N chunks, of which any subset of $N - 2f$ chunks will decode into B. C already satisfies this property, and the `Encode` process is deterministic, so it must be `Encode`$(B) = C$, and the lemma is satisfied in this case.

Case 2: Inconsistent encoding. Assume there exist two subsets D_1, D_2 of $N - 2f$ chunks in C, and `Decode`$(D_1) \neq $`Decode`$(D_2)$. Let `Decode`$(D_1) = B_1$

and `Decode`$(D_2) = B_2$ where $B_1 \neq B_2$. We want to prove that for any subset D of $N - 2f$ chunks in C, `MerkleRoot`$($`Encode`$($`Decode`$(D))) \neq $`MerkleRoot`$(C)$.

We prove it by showing there does *not* exist any block B such that $C = $`Encode`$(B)$. That is, C is not a consistent encoding result of any block. Assume for contradiction that there exists B' such that $C = $`Encode`$(B')$. Because D_1 is a subset of $N-2f$ chunks in C and `Decode`$(D_1) = B_1$, it must be $B_1 = B'$, otherwise the semantic of erasure code is broken. For the same reason $B_2 = B'$, so $B_1 = B_2$. However it contradicts with $B_1 \neq B_2$, so the assumption must not hold, and there does not exist any block B such that $C = $`Encode`$(B)$.

We now prove that `MerkleRoot`$($`Encode`$($`Decode`$(D))) \neq $`MerkleRoot`$(C)$ for any subset D of $N - 2f$ chunks in C. Assume for contradiction that `MerkleRoot`$($`Encode`$($`Decode`$(D))) = $`MerkleRoot`$(C)$, then it must be that $C = $`Encode`$($`Decode`$(D))$ because Merkle root is a secure summary of the chunks. This contradicts with the result we have just proved: there does not exist any block B such that $C = $`Encode`$(B)$. So the assumption cannot hold, and the lemma is satisfied in this case. □

Theorem B.9 (Correctness)**.** *If a correct server has `Completed`, then correct clients always reconstruct the same block B' by invoking `Retrieve`. Also, if a correct client initiated the dispersal by invoking `Disperse`(B) and no other client invokes `Disperse`, then $B = B'$.*

Proof. We first prove the first half of the theorem: any two correct clients always return the same data upon finishing `Retrieve`. By Lemma B.7, any two clients will set their `ChunkRoot` to the same value. Note that a client sets `ChunkRoot` to the root of the chunks it uses for decoding. This implies that any two correct clients will use subsets from the *same* set of chunks. By Lemma B.8, either:
1. They both decode and obtain the same block B'.
2. Or, they each compute `MerkleRoot`$($`Encode`$())$ on the decoded block and both get a result that is different from `ChunkRoot`.

In the first situation, both clients will return B'. In the second situation, they both return the block containing string "BAD_UPLOADER". In either case, they return the same block.

We then prove the second half of the theorem. Assume a correct client has initiated `Disperse`(B) and no other client invokes `Disperse`. By Theorem B.6, any correct client invoking `Retrieve` will obtain some block B'. We now prove that $B' = B$. Assume for contradiction that $B' \neq B$. Then the client must have received $N - 2f$ `ReturnChunk`$($`MerkleRoot`$($`Encode`$(B')), \cdot, \cdot)$ messages. At least one of them must come from a correct server because $N - 2f > f$, so at least one correct server have `ChunkRoot` set to `MerkleRoot`$($`Encode`$(B'))$. However, because there is only invocation of `Disperse`(B), all correct servers must have set `ChunkRoot` to `MerkleRoot`$($`Encode`$(B))$.

Phase 1. Dispersal at the i-th server

1. For $1 \le j \le N$, let $V_i^e[j]$ be the largest epoch number t such that $\text{VID}_j^1, \text{VID}_j^2, ..., \text{VID}_j^t$ have Completed.
2. Let B_i^e be the block to disperse (propose) for epoch e. B_i^e contains two parts: transactions T_i^e and observation V_i^e.
3. Invoke $\text{Disperse}(B_i^e)$ on VID_i^e as a client.
- Upon Complete of VID_j^e ($1 \le j \le N$), if we have not invoked Input on BA_j^e, invoke $\text{Input}(1)$ on BA_j^e.
- Upon $\text{Output}(1)$ of least $N-f$ BA instances, invoke $\text{Input}(0)$ on all remaining BA instances on which we have not invoked Input.
- Upon Output of all BA instances,
  1. Let local variable $S_i^e \subset \{1...N\}$ be the indices of all BA instances that $\text{Output}(1)$. That is, $j \in S$ if and only if BA_j^e has $\text{Output}(1)$ at the i-th server.
  2. Move to retrieval phase.

Phase 2. Retrieval

1. For all $j \in S_i^e$, invoke Retrieve on VID_j^e to download full block $B_j^{e\prime}$. Decompose $B_j^{e\prime}$ into transactions $T_j^{e\prime}$ and observation $V_j^{e\prime}$. Let $V_j^{e\prime} = [\infty, \infty, ..., \infty]$ if $B_j^{e\prime}$ is ill-formatted.
2. Deliver $\{T_j^{e\prime} | j \in S_i^e\}$ (sorted by increasing indices). Set $\text{Delivered}[e][j] = 1$ (initially 0) for all $j \in S_i^e$.
3. For $1 \le j \le n$, let $E_i^e[j]$ be the $(f+1)^{th}$-largest value among $\{V_k^{e\prime}[j] | k \in S_i^e\}$.
4. For all $1 \le j \le N$, for all $1 \le d \le E_i^e[j]$, check if $\text{Delivered}[d][j] = 0$. If so, invoke Retrieve on VID_j^d to download full block $B_j^{d\prime}$, and set $\text{Delivered}[d][j] = 1$ (initially 0).
5. Deliver all blocks downloaded in step 4 (sorted by increasing epoch number and node index).

Figure 17: Algorithm for DispersedLedger with inter-node linking. The blue color indicates the changes from the single-epoch algorithm.

So $\text{MerkleRoot}(\text{Encode}(B)) = \text{MerkleRoot}(\text{Encode}(B'))$ This contradicts with our assumption, so the assumption must not hold, and $B = B'$. □

C Specification of the full DispersedLedger protocol with Inter-node Linking

Figure 17 describes how to modify the single-epoch protocol to use inter-node linking. Blue color highlights the parts are added compared to the single-epoch protocol.

D Correctness proof of DispersedLedger

Notations. Let H ($H \subset \{1,2,...,N\}$) be the set of the indices of correct nodes. That is, $i \in H$ if and only if the i-th node is correct. In our proof, we use the variables in the full algorithm defined in Fig. 17. We also use "phase x, step y" to refer to specific steps in Fig. 17.

Lemma D.1. *For any epoch e, any $i \in H$, and any $1 \le j \le N$, if $j \in S_i^e$ then VID_i^e has Completed at some correct node.*

Proof. By the definition of S_i^e (phase 1, step 3), $j \in S_i^e$ if and only if BA_j^e has $\text{Output}(1)$ at the i-th node. By the Validity property of BA (§4.1), BA_j^e $\text{Output}(1)$ at a correct node implies that at least one correct node has invoked $\text{Input}(1)$ on BA_j^e, which only happens when that node sees VID_j^e Complete (phase 1, step 3). □

Lemma D.2. *For any epoch e, any $i, j \in H$, $S_i^e = S_j^e$ and $E_i^e = E_j^e$.*

Proof. By the definition of S_i^e (phase 1, step 3), $k \in S_i^e$ if and only if BA_k^e has $\text{Output}(1)$ at the i-th node. By the Agreement property of BA (§4.1), BA_k^e will eventually $\text{Output}(1)$ at the j-th node. So $k \in S_i^e$ if and only if $k \in S_j^e$, and $S_i^e = S_j^e$.

We now prove $E_i^e = E_j^e$. The i-th node (which is correct) starts the computation of E_i^e by invoking Retrieve on all VIDs in $\{\text{VID}_k^e | k \in S_i^e\}$. These Retrieves are guaranteed to finish by Lemma D.1 and the Availability property of VID (Theorem B.6). The node then extracts the observations $\{V_k^{e\prime} | k \in S_i^e\}$ from the downloaded blocks. Note that the j-th node will download the same set of observations. This is because $S_i^e = S_j^e$, and the VID Correctness property (Theorem B.9) guarantees the j-th node will obtain the same blocks when invoking Retrieve on $\{\text{VID}_k^e | k \in S_j^e\}$.

To combine the observations into the estimation, the i-th node runs phase 2, step 3. This process is deterministic, with E_i^e being a function of the observations $\{V_k^{e\prime} | k \in S_i^e\}$ and parameter f. Because we have just proved the j-th node will obtain the same set of estimations, and by our security model f is a protocol parameter known to all nodes (§2.4), the j-th node will get the same results.

□

Lemma D.3. *For any epoch e, and any $i \in H$, $|S_i^e| \ge N - f$. That is, S_i^e contains at least $N-f$ indices.*

Proof. By the definition of S_i^e (phase 1, step 3), this lemma essentially states that at least $N - f$ BAs among $\{\text{BA}_1^e, \text{BA}_2^e, ..., \text{BA}_N^e\}$ will $\text{Output}(1)$ at the i-th node.

Assume for contradiction that $|S_i^e| < N - f$. By Lemma D.2, $|S_j^e| < N - f$ for all $j \in H$, i.e., less than $N - f$ BAs eventually $\text{Output}(1)$ at every correct node. We now consider the possible outcomes of the remaining BA instances, which do not eventually $\text{Output}(1)$.

One possibility is some of them $\text{Output}(0)$. According to phase 1, step 3, correct nodes will not invoke $\text{Input}(0)$ on any BA instance unless $N - f$ BA instances have $\text{Output}(1)$. By our assumption, less than $N - f$ BA $\text{Output}(1)$, so the latter is not happening and *no* correct nodes will $\text{Input}(0)$ on *any* BA instance. By the Validity property of BA (§4.1), no BA instance can $\text{Output}(0)$.

We have showed that the remaining BAs cannot $\text{Output}(0)$, so it must be that all of them never terminate. We will prove it is also impossible. Assume for contradiction that all BAs that do not $\text{Output}(1)$ never terminate. By our assumption,

less than $N-f$ BAs $\texttt{Output}(1)$, so there must exist $k \in H$ such that $\texttt{BA}_k^e$ never terminates. By the Termination property of VID (Theorem B.2), $\texttt{VID}_k^e$ eventually $\texttt{Completes}$ on all correct nodes. According to phase 1, step 3, because not all BAs will terminate, all correct nodes will stay at this step. All correct nodes will $\texttt{Input}(1)$ to $\texttt{BA}_k^e$ upon seeing $\texttt{VID}_k^e$ $\texttt{Complete}$. By the Termination and Validity properties of BA (§4.1), $\texttt{BA}_k^e$ will terminate and $\texttt{Output}(1)$, which conflicts with our assumption.

We have showed there is no valid outcome for the remaining BA instances, so our assumption cannot hold, and at least $N-f$ BA instances eventually $\texttt{Output}(1)$ at all correct nodes. $\qquad \square$

Lemma D.4. *For any epoch e, any $i \in H$, and any $1 \leq j \leq N$, there exist $p,q \in H$ such that $V_p^e[j] \leq E_i^e[j] \leq V_q^e[j]$.*

Proof. The lemma states that if the i-th node (which is correct) computes the estimation $E_i^e[j]$ for the j-th node, then the estimation is lower- and upper-bounded by the *observations* $V_p^e[j]$ and $V_q^e[j]$ of two correct nodes (with indices p and q). That is, the estimation is not too high or too low.

Now assume for contradiction that for some $1 \leq j \leq N$, for all $p \in H$, $V_p^e[j] > E_i^e[j]$. That is, the estimation for j is not lower bounded by the observations made by any correct node. According to phase 2, step 3, the i-th node sets $E_i^e[j]$ to the $(f+1)^{th}$-largest value among $\{V_k^{e\prime}[j]|k \in S_i^e\}$. Here, $V_k^{e\prime}$ is the observation of the k-th node downloaded by invoking $\texttt{Retrieve}$ on $\texttt{VID}_k^e$. By Lemma D.1 and VID Availability property (Theorem B.6), the $\texttt{Retrieves}$ will eventually finish.

By our assumption, for all $p \in H \cap S_i^e$, $V_p^e[j] > E_i^e[j]$. By the VID Correctness property (Theorem B.9), the observations of correct nodes will be correctly downloaded. That is, $V_k^{e\prime} = V_k^e$ for all $k \in H$. So for all $p \in H \cap S_i^e$, $V_p^{e\prime}[j] > E_i^e[j]$. By Lemma D.3, $|S_i^e| \geq N-f$, so $|H \cap S_i^e| \geq N-2f$. So there are *at least* $N-2f$ values in $\{V_k^{e\prime}[j]|k \in S_i^e\}$ that are greater than $E_i^e[j]$. However, $E_i^e[j]$ is the $(f+1)^{th}$-largest value among $\{V_k^{e\prime}[j]|k \in S_i^e\}$, so there can be *at most* f values in $\{V_k^{e\prime}[j]|k \in S_i^e\}$ that are greater than $E_i^e[j]$. Because $N > 3f$ (§2.4), $N-2f > f$, so the two conclusions are in conflict, and the assumption cannot hold.

We can similarly prove it is impossible that for some $1 \leq j \leq N$, for all $q \in H$, $V_q^e[j] < E_i^e[j]$. $\qquad \square$

Theorem D.5 (DispersedLedger is well-defined). *For any epoch e, any $i \in H$, the i-th node eventually finishes epoch e.*

Proof. This lemma states that correct nodes will never be stuck in any epoch e, so that our algorithm is well-defined. To prove that, we go through Fig. 17 line by line and prove each step will eventually finish.

Phase 1, steps 1–2. These are local computation and will finish instantly.

Phase 1, step 3. This step finishes as soon as all BA instances in that epoch $\texttt{Output}$. By Lemma D.3, all correct nodes eventually see at least $N-f$ BA instances $\texttt{Output}(1)$. At that point, each correct node will invoke $\texttt{Input}(0)$ into all BAs on which it has not invoked $\texttt{Input}$. This ensures that all correct nodes eventually invoke $\texttt{Input}$ on all BAs. By the Termination property of BA (§4.1), all BAs will eventually $\texttt{Output}$ on all correct nodes, which ensures this step will finish.

Phase 2, step 1. This step finishes as soon as $\texttt{Retrieves}$ on $\{\texttt{VID}_j^e|j \in S_i^e\}$ finish. By Lemma D.1, $\{\texttt{VID}_j^e|j \in S_i^e\}$ will $\texttt{Complete}$ on all correct nodes. Then by VID Availability property (Theorem B.6), the $\texttt{Retrieves}$ will finish.

Phase 2, steps 2–3. These are local computation and will finish instantly.

Phase 2, step 4. This step will finish if for all $1 \leq j \leq N$, for all $1 \leq d \leq E_i^e[j]$, $\texttt{Retrieve}$ of $\texttt{VID}_j^d$ finishes. By Lemma D.4, there exists $q \in H$ such that $V_q^e[j] \geq E_i^e[j]$, and the q-th node (which is correct) reports that $\texttt{VID}_j^t$ has $\texttt{Completed}$ for all $1 \leq t \leq V_q^e[j]$. By VID Availability property (Theorem B.6), the $\texttt{Retrieves}$ will eventually finish, so this step will finish.

Phase 2, steps 5. This is local computation and will finish instantly. $\qquad \square$

Theorem D.6 (Validity). *All blocks proposed by correct nodes are eventually delivered by all correct nodes.*

Proof. Assume the i-th node (which is correct) proposes block B_i^e in epoch e. The i-th node invokes $\texttt{Disperse}(B_i^e)$ on $\texttt{VID}_i^e$. By VID Termination property (Theorem B.2), eventually all correct nodes will see $\texttt{VID}_i^e$ $\texttt{Complete}$. So there must exist an epoch t where for all $j \in H$, $V_j^t[i] \geq e$. That is, in epoch t, all correct nodes report that the i-th node has at least dispersed into $\texttt{VID}_i^1$ to $\texttt{VID}_i^e$. By Lemma D.4, for all $j \in H$, $E_j^t[i] \geq e$. According to phase 2, steps 4–5, all correct nodes either have already delivered B_i^e in previous epochs, or will deliver B_i^e in epoch t. $\qquad \square$

Theorem D.7 (Agreement and Total Order). *Two correct nodes deliver the same sequence of blocks.*

Proof. Let $i,j \in H$. We prove this theorem by induction on the number of epochs the i-th and the j-th nodes have finished. In other words, we prove that for any $t \geq 0$, the i-th and the j-th nodes deliver the same sequence of blocks in the first t epochs.

Initial ($t = 0$). Both nodes have not delivered any block. So the hypothesis clearly holds in this situation.

Induction step. Assume our hypothesis holds for $t = e-1$ ($e \geq 1$). We now prove the hypothesis holds for $t = e$. We first show the two nodes commit the same sequence of blocks with BA. By Lemma D.2, $S_i^e = S_j^e$ and $E_i^e = E_j^e$. According to phase 2, step 1, both nodes will invoke $\texttt{Retrieve}$ on the same set of VIDs. By VID Correctness property (Theorem B.9), they will get the same set of blocks and deliver them in the same order in phase 2, step 2.

We now show the two nodes commit the same sequence of blocks with inter-node linking. The local variable $\texttt{Delivered}$

stores whether a node has delivered a block (phase 2, steps 2, 4). By the induction hypothesis, the two nodes have delivered the same sequence of blocks prior to epoch e, so the variable `Delivered` is the same on the two nodes. By Lemma D.2, $E_i^e = E_j^e$. So the two nodes will invoke `Retrieve` on the same set of VIDs in phase 2, step 4 and get the same set of blocks. Both nodes sort the blocks deterministically and deliver them in the same order in phase 2, step 5.

We have proved that the i-th and the j-th nodes deliver the same sequence of blocks in epoch e. By our induction hypothesis, they deliver the same sequence until epoch $e-1$. So they deliver the same sequence in the first e epochs. This completes the induction. $\qquad\square$

Re-architecting Traffic Analysis with *Neural* Network Interface Cards

Giuseppe Siracusano
NEC Laboratories Europe

Salvator Galea
University of Cambridge

Davide Sanvito
NEC Laboratories Europe

Mohammad Malekzadeh
Imperial College London

Gianni Antichi
Queen Mary University of London

Paolo Costa
Microsoft Research

Hamed Haddadi
Imperial College London

Roberto Bifulco
NEC Laboratories Europe

Abstract

We present an approach to improve the scalability of online machine learning-based network traffic analysis. We first make the case to replace widely-used supervised machine learning models for network traffic analysis with *binary neural networks*. We then introduce **N**eural **N**etworks on the **NIC** (N3IC), a system that compiles binary neural network models into implementations that can be directly integrated in the data plane of SmartNICs. N3IC supports different hardware targets, and it generates data plane descriptions using both micro-C and P4 languages.

We implement and evaluate our solution using two use cases related to traffic identification and to anomaly detection. In both cases, N3IC provides up to a 100x lower classification latency, and 1.5-7x higher throughput than state-of-the-art software-based machine learning classification systems. This is achieved by running the entire traffic analysis pipeline within the data plane of the SmartNIC, thereby completely freeing the system's CPU from any related tasks, while forwarding traffic at line rate (40Gbps) on the target NICs. Encouraged by these results we finally present the design and FPGA-based prototype of a hardware primitive that adds binary neural network support to a NIC data plane. Our new primitive requires less than 1-2% of the logic and memory resources of a VirteX7 FPGA. We show through experimental evaluation that extending the NIC data plane enables more challenging use cases that require online traffic analysis to be performed in a few microseconds.

1 Introduction

Online traffic analysis is a fundamental building block in today's networks, as it enables traffic classification [2, 5, 14, 26], security [10, 25, 31] and application-specific traffic forwarding strategies [40]. The complexity of network traffic patterns and the use of encrypted communications are driving the widespread adoption of traffic analysis based on Machine-Leaning (ML), implemented on commodity servers [13]. However, it is challenging to meet the throughput and latency requirements of modern networks while performing ML-based traffic

analysis [47]. Current high-performance solutions use programmable network interface cards (NICs) [12, 29, 48] to offload parts of the traffic analysis (e.g., flow statistic collection [1, 3, 28]) directly in their data plane, while still performing machine learning inference on a separate executor, e.g., the host's CPU. Unfortunately, moving the collected flow statistics across sub-systems introduces an important bottleneck [30], forcing high throughput solutions to send collected data to the ML executor in batches, thus sensibly increasing the processing latency (§ 2).

Recognizing that running ML inference *within* the network data plane would avoid data movements and solve the issue, state-of-the-art solutions implement widely used techniques, i.e., Decision Trees and their ensembles (Random Forests), using match-action tables, which are available within a NIC data plane [8, 55]. However, these solutions rely on expensive TCAM memories, and fitting Decision Trees in match-action tables requires restricting their depth to a few levels, thus impacting their accuracy. More specifically, [55] reports a maximum of five levels implemented on the NetFPGA, while [8] supports only Decision Trees of depth four on the Barefoot Tofino. Therefore, currently, network operators have to compromise between throughput, latency, or accuracy.

In this paper, we propose a new approach that efficiently leverages programmable NICs' hardware (and can achieve high throughput and low latency) while maintaining comparable accuracy with respect to existing ML-based traffic-analysis solutions implemented in software. The key insight is to exploit binary neural networks (BNNs) [15], a recently-proposed ML model targeting battery-powered edge devices. We show that BNNs can provide better classification accuracy than Decision Trees and Random Forests on the tested traffic analysis tasks (§ 3). Importantly, BNNs use single bits to represent inputs and weights, which provides two critical properties: (i) they exhibit a very compact memory footprint even for larger models; (ii) unlike mainstream Deep Neural Networks (DNNs), BNNs require only simple operations such as XOR and population count. This enables the implementation of efficient BNNs executors in a NIC's data plane,
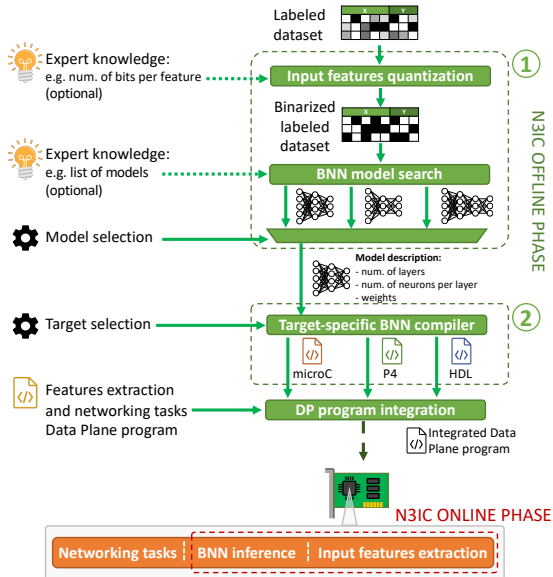
Figure 1: N3IC overview. Users provide a labeled dataset. N3IC uses it to generate a binary neural network model, which is then compiled to a data plane program for a target NIC.

without requiring expensive resources, such as TCAMs.

Building on this insight, we developed N3IC, a complete solution to perform network traffic analysis using BNNs with commodity programmable NICs. N3IC comprises two key components (Figure 1, § 4): ① a framework to train a BNN using a labeled dataset provided by the user, and ② a compiler that translates the trained model into target-specific executable code. To show the generality of our approach, we implement two compiler backends: one targeting micro-C, a subset of the C language used by Netronome SoC-based NICs [29], and one targeting the P4 language [6]. The latter enables compiling to a growing set of P4-enabled NICs [21], including FPGA-based NICs using the P4->NetFPGA toolchain [16].

Furthermore, we evaluate the cost of providing BNN execution as a *native* hardware primitive that can be exposed to high-level programming languages (e.g., using P4's *extern*). We prototype this on the NetFPGA using RTL description language and show it only needs a modest 1-2% of a Xilinx Virtex7 FPGA's logic resources. While prior work has shown the potential of implementing ML models on FPGA [24, 51], they target ML models for application-level data processing, which has millisecond-scale latency requirements (as opposed to microsecond), and they are typically based on FPGA monolithic implementations. To the best of our knowledge, we are the first ones integrating a streamlined BNN executor, tailored to network traffic analysis models, *within* the NIC data plane.

We evaluate N3IC across different hardware platforms using traffic classification, security anomaly detection and network tomography as use cases (§ 6). Results show that N3IC can perform traffic analysis with high accuracy and with la-

tency in the microseconds, for millions of network flows per second, while processing packets at NICs' line rate. Compared to a similar system that implements the traffic analysis on a general-purpose CPU (with packet forwarding and feature extraction still offloaded to the NIC), N3IC provides up to 7x higher throughput and up to 100x lower latency.

Contributions. In this paper, we:

- demonstrate that BNNs provide high accuracy and low memory footprint for the selected traffic analysis use cases.
- design and implement an end-to-end system that performs traffic analysis in programmable NICs' data plane: this includes a framework to train BNNs and a compiler that translates models into both P4 and Netronome's micro-C.
- develop a new hardware primitive that enables BNN inference as first-class-primitive for next-generation programmable NICs.
- evaluate our solution on three traffic analysis use cases: (i) traffic classification, (ii) anomaly detection, and (iii) network tomography.
- Source code to reproduce key results of our work is at: https://github.com/nec-research/n3ic-nsdi22

2 Motivation and Challenges

Motivation. Modern data-center networks comprise a variety of network appliances, e.g., traffic classifiers, load balancers, and security middleboxes [34, 36]. They need to handle over a million of flows per second while only incurring a few tens of microseconds of processing latency per packet to avoid affecting the end-to-end latency [11, 23].

To meet these tight requirements, mainstream systems offload the packet capture and feature extraction steps to a programmable NIC [1, 28]. Periodically, the host system polls the extracted features from the NIC, and performs the analysis step. This approach relieves the load on the host's CPU and achieves higher throughput but at the cost of higher processing latency. To illustrate this trade-off in practice, we set up an experiment in which we offload the feature extraction on the Netronome NFP4000 NIC while we execute the analysis on an Intel E5-1630 v3 CPU. The results in Figure 2 (*NIC+CPU* line) show that as the throughput increases, the processing latency scales super-linearly. For instance, at 0.2M flows per second, the latency is 42μs but if we increase the throughput to 1M flows per second, the latency grows beyond 800μs.

There are two reasons for this. First, having the feature extraction and analysis steps running on two different subsystems requires moving data, e.g. crossing the PCIe bus, which can take up to a few microseconds [30]. Second, and most critically, CPUs require input data batching to improve the per-core processing efficiency. Batching improves data locality, avoiding stalls in the CPU pipeline due to data read delay, and it allows to fill the CPU's vector processing registers, thereby increasing the overall throughput but at the expense of much higher latency. This trade-off also applies to GPUs, which extensively rely on batching to achieve high through-
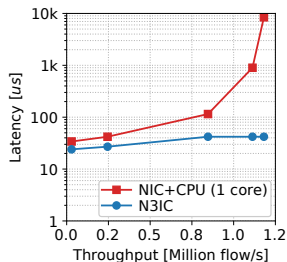
Figure 2: Processing latency when increasing throughput (flows per second) for a baseline performing feature extraction on the NIC and analysis on the CPU (*NIC+CPU*) and our system `N3IC`.
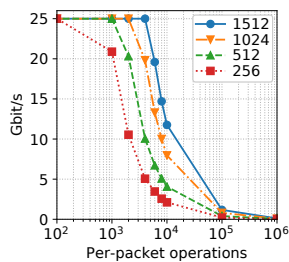

Figure 3: Forwarding throughput on a Netronome NFP4000 NIC, for different packet sizes when increasing the number of operations per packet.

put, and it explains why even network-attached GPUs [32] are not well-suited for low-latency packet processing.

A way to address the above issues is to perform the analysis directly within the subsystem that collects the data to be analyzed, i.e., within the NIC data plane. This would allows us to *(i)* avoid data movements from one subsystem to the other, and *(ii)* leverage the architectures of programmable NICs tailored to perform latency-efficient per-packet processing. As we detail in the rest of the paper, this indeed enables maintaining low latency (<40us) even at high throughput, as shown by the performance of `N3IC` in Figure 2.

Challenges. Existing solutions advocating for performing ML inference in the data plane of packet-processing hardware [8, 55] strictly rely on match-action tables that support ternary-matching. These resources are *(i)* not always available in a NIC data plane; *(ii)* costly, when available, since they use ternary content-addressable memory (TCAM), which is about 6x more expensive in terms of silicon area than SRAM [7]; *(iii)* limited; thus enabling only Decision Trees with small depth with an impact on the inference accuracy.[1] While using exact-matching tables may be a workaround, it would require enumerating the values to match on. For instance, to handle a single 16b feature, we may need to add 65k entries.

Enabling ML inference without the use of match-action tables resources and doing so while guaranteeing high-throughput, low latency, and high-accuracy requires solving three key challenges. First, existing programmable NICs have at most few 10s of MBs of fast on-chip SRAM memory [29, 48, 49]. Most of this memory, though, is needed to store forwarding and policy tables, leaving little space available for application data. This makes it hard to implement ML models within the NIC, often requiring trading-off model complexity for memory utilization. Second, to achieve high throughput the application logic needs to be highly parallelizable in order to fully utilize all compute resources on a NIC.

---

[1] Both IISY [55] and pforest [8] report an ability to run Decision Tree models with depth capped to five and four layers, respectively.

In fact, the NIC may provide a good amount of available processing resources if its architecture parallelism is leveraged. We show this in Figure 3, which plots the throughput achieved on the Netronome NFP4000 SmartNIC for different packet sizes as we increase the number of operations performed per packet. The larger the average packet size (and, hence, the less packets per second need to be processed), the higher is the number of operations that can be performed, before the forwarding throughput is negatively impacted. Finally, some ML models require complex arithmetic functions, such as multiplications or floating-point operations, which usually are not available on programmable NICs [45]. This limitation does not only affect the implementation of the ML model, but it also impacts the ability to perform pre-processing on the input features, as required by some models such as Support-Vector-Machine or K-Nearest-Neighbor.

3 Traffic Analysis with BNN

In this section, we show that binary neural networks (BNNs) are a promising option to address these challenges. Originally proposed for energy-efficient image processing on battery-powered devices, BNNs are an extreme quantized version of traditional DNNs in which each weight is encoded in just one bit rather than the typical 8-, 16- or 32-bit values. This makes them particularly appealing for our goals due to the following reasons. First, the single-bit input and weights drastically reduce their memory footprint. Second, the BNN's neurons perform a XOR between the input and weight vectors, and use as activation function the sign function on the population count (`popcnt`) performed on the bit vector resulting from the XOR. Therefore, they can be implemented efficiently (and with high performance) in hardware since XOR and `popcnt` operations are commonly supported by most platforms.

Unsurprisingly, for complex tasks such as image recognition, BNNs exhibit 3-10% points lower prediction accuracy than fully-fledged DNNs [20]. However, as we illustrate in the rest of this section, network traffic analysis models are usually much simpler and this enables BNNs to achieve an accuracy comparable (if not better) than existing implementations relying on decision trees and random forests.

3.1 Use cases

We introduce two typical traffic analysis uses cases that we use as running examples throughout the paper: IoT Traffic Classification and Security Anomaly Detection. Both use cases are general machine learning classification tasks, and therefore they are representative of common analysis use cases performed on network traffic. Further, they have open datasets, which helps making our results reproducible.

IoT Traffic Classification assigns an IoT device type to an observed network flow. For instance, this can be used in edge networks by operators to assign IoT traffic to specific Quality-of-Service classes. We focus on a 10-classes classification
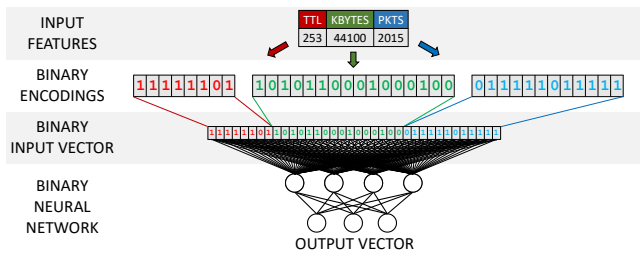
Figure 4: The binary input vector is a concatenation of features' binary encodings. Each feature is represented using the minimum required number of bits to represent its values range. The number of bits per feature has to be fixed at training time.

task, where each flow is assigned to 9 possible device categories, such as home assistants, IoT cameras, sensors, or to a 10-th class that includes *anything else*, e.g., smartphones or laptops network traffic. We use 17 flow-level features to perform the classification. Examples of features are the number of packets and bytes being transferred, the mean packet interarrival time or the mean packet sizes. We remark that the selected features are not specific to this use case: they are widely supported in open source tools and used in production settings [35]. We use the dataset published by [44].

Security Anomaly Detection is about flagging network flows that are related to security issues, such as Denial-of-Service attacks, port scans, etc. This is a network analysis task widely applied in networks of any size, and in different scenarios including telecom operators networks, datacenters, and enterprises. For this task, flows are classified into two classes, i.e., good or bad. Usually, this kind of classification is used to potentially trigger more expensive downstream analysis on the traffic, and it has the goal to capture the large bulk of potentially malicious network interactions, rather than guaranteeing complete protection. For this task, in addition to the 17 flow-level features reported earlier, we add 3 additional features that look at the behavior of multiple flows. For instance, we consider the number of flows from a single source IP address. Like in the previous case, these features are well-known and widely adopted in operational settings. We use the dataset published by [27].

3.2 BNN Analysis Pipeline

To apply BNNs, we have to define the input features quantization strategy, to convert float and integer numbers into BNN's binary features. Then, we perform the training of the binary neural network using the labeled dataset. Finally, we evaluate the classification performance with previously unseen data.

Input preparation Previous work on BNNs introduces a first regular non-binary network layer that is trained together with the remaining binarized layers. This enables "learning" the quantization strategy for the features, but at the same time, it introduces multiplication operations within the first layer. We cannot afford to perform such operations in the data plane. Therefore, we designed a different quantization approach (Fig-

ure 4): we use as input to the BNN the concatenation of the flow feature values' binary representations. For instance, an input feature in the range 0-255 can be represented by an 8b vector. Our approach has two advantages: it does not require any additional processing since we reuse the hardware representation of the features; and it allows to assign to the features the number of bits their value ranges require. For instance, a single vector of 64b can be used to represent 4 features on 16b, or 3 features on 16b and 2 features on 8b; and so on.

BNN Training Like other ML models, BNNs need to be trained offline on a training dataset, in order to define the values of the weights that will be used during the online analysis phase. We perform training using the technique from Courbariaux and Bengio [9], which is based on a canonical back-propagation algorithm. This solution trains the network using float values, but it ensures that the BNN's weights converge to values included in the [-1, 1] range, and that they are normally distributed around 0. This helps in reducing the loss of information when the float weight values are mapped to just two values, i.e., 0 and 1 [9].

BNN traffic analysis performance We test three different BNNs architectures, each with 256 input binary features and three fully-connected layers. The three models differ by the number of neurons in the hidden layers: [32, 16, 10]; [64, 32, 10]; [128, 64, 10]. For both datasets, we use a 256b input vector. Although we have 17 and 20 features for the two cases, respectively, we can represent different number of features with the same binary input vector size by changing the number of bits used to represent each feature, as mentioned earlier (cf. Figure 4). We compare the BNNs to Decision Trees (DT) and Random Forests (RFs). For DTs, we vary the depth of the tree, between 3 and 10. RFs are an ensemble of DTs, therefore they have as an additional hyperparameter the number of trees, which we vary between 3 and 5. For readability, since the trends are similar, we only plot a subset of the results in the figures, i.e., three depth values of 3, 6, and 9, and always 5 trees for the RF. In all the tests, we perform 5-fold cross-validation, and report averaged results.[2]

In Figure 5 and Figure 6 we plot the classification accuracy vs the amount of memory required by the ML models, for the IoT and Security use cases, respectively. In the top plots, we do not make distinction between memory of type SRAM, used by BNN implementations, and of type TCAM, required by DT and RF implementations. Here, we can observe that the two larger BNNs achieve an accuracy that is closer to that of DTs and RFs of at least depth 6. The two larger BNNs achieve 96% and 97.4% using 2.5KB and 5.5KB of SRAM, vs 97% and 96.9% accuracy of DT6 and RF6, using 1.3KB and 6.4KB of TCAM, respectively. The smaller BNN achieves 92.4% accuracy using 1.2KB of SRAM. In the Security dataset, the classification is harder, and only the

---

[2]The IoT dataset is balanced across the 10 classification categories, with each category having 43k distinct flows. In the Security dataset, we have a binary classification with 164k anomalous and 90k normal flows.
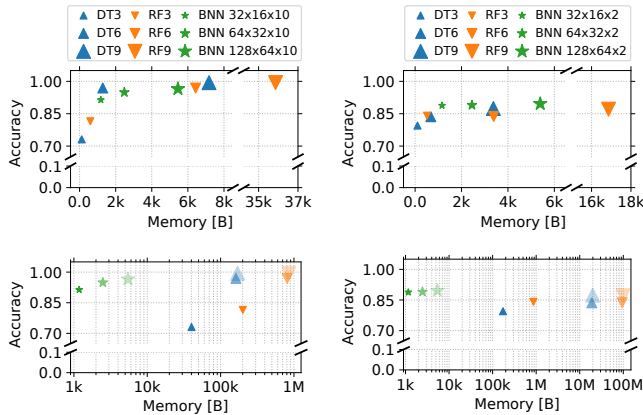
Figure 5: Accuracy vs Memory (bytes) scatter plot for DT and RF on the IoT dataset. BNNs always use SRAM-based implementations. DTs and RFs use TCAM (top) and SRAM (bottom) implementations.
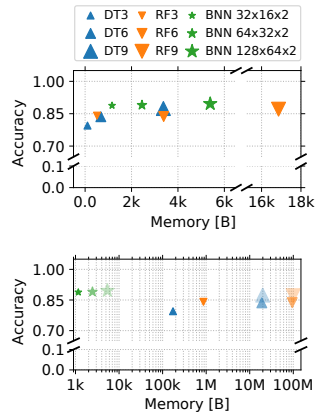
Figure 6: Accuracy vs Memory (bytes) scatter plot for DT and RF on the Security dataset. BNNs always use SRAM-based implementations. DTs and RFs use TCAM (top) and SRAM (bottom) implementations.
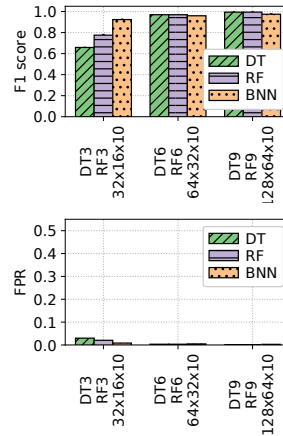
Figure 7: F1 score (top) and FPR (bottom) of BNN models on the IoT dataset, compared to DTs and RFs.

Figure 8: F1 score (top) and FPR (bottom) of BNN models on the Security dataset, compared to DTs and RFs

larger DT9 and RF9 achieve accuracy above 90%, using respectively 3.4KB and 16.9KB of TCAM. The smallest BNN achieves 91.1% accuracy using just 1.2KB of SRAM. However, it should be noted that ternary matching with TCAM is roughly 6x more expensive than binary matching with SRAM, in terms of required silicon resources [7] and TCAM is often not available on NICs.

SRAM implementations: To compare the memory requirements when targeting similar hardware, in the bottom plots of Figure 5 and Figure 6 we show the memory consumption of DTs and RFs when using SRAM-based implementations. As described in [55], in the absence of TCAM support from the hardware target, all the values of the features selected by model fitting have to be enumerated as appearing in the data. Given that some of our features are flow statistics, the values they can potentially assume range from the minimum to the maximum observed from the data. In fact the memory requirements for DTs and RFs grow orders of magnitude larger (in this case, the plots have the x axis in log scale). Even the smallest DT3 model requires at least 40.2KB of SRAM for the IoT case, and 173.3KB for the Security case.

F1-score and FPR: We now look more carefully at the classifier performance, reporting F1-score and False Positive Rate (FPR) for the tested models. The F1-score is a harmonic mean of Precision and Recall, whereas the False Positive Rate tells the quota of negative samples mis-classified as positive, in a two-classes classifier. For this metric, in the IoT case that has 10 classes, we use a 1-vs-all strategy. The BNN models achieve always better F1-score when compared to the smallest DT and RF models, in both use cases. For larger models, the F1-score is in the range 88-91.6 in all cases, showing
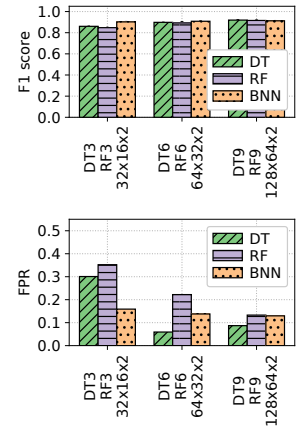
relatively small variations among classifiers.

For the FPR, it is important to consider this metric in relation to the *Recall* of the classifier. In fact, a low FPR may be a symptom of a classifier assigning very few samples to the positive class. We can see this in Figure 7. DT3 and RF3 appear to have a relatively good FPR (3.0% and 2.1%). However, these low FPRs are due to the classifiers inability to identify the positive class. In particular, as captured also by F1-scores, DT3 and RF3 have a low Recall of 73.1% and 81.5% for the IoT case, whereas the smallest BNN has Recall at 92.4% with an FPR of 0.8%. We can see a similar issue in the Security use case (Figure 8). For instance, DT6 has FPR at 5.9% but Recall at 88.2%, whereas the smallest BNN has a higher FPR of 15.9% with a better Recall at 95.1%. Here, it should be noted that in this use case a reasonably higher FPR is not necessarily an issue. The anomaly detection is often used as a filter, before performing more expensive analysis on the flows classified as suspicious, e.g., diverting the traffic to a Scrubbing Center [33]. We provide more results in Appendix.

4 System design and implementation

We now present the design and implementation of N3IC, our end-to-end solution that enables to perform traffic analysis within a NIC data plane using BNNs (cf. Figure 1).

N3IC operations N3IC takes a training labeled dataset as input, and outputs programs that can be integrated into a target device's data plane. Currently, we support outputs in micro-C and P4 languages, targeting SoC-based Netronome SmartNICs and PISA-based architectures, respectively. N3IC entirely automates the generation of the BNN model and its implementation in the target data plane programming language. However, programmers need to perform the final integration step, to connect the input features extracted from the network packets with the programs generated by N3IC. In fact, feature extraction may happen in different ways, and

Figure 9: The architecture of a Netronome NFP4000's programmable blocks and the BNN processing with `N3IC`-NFP.

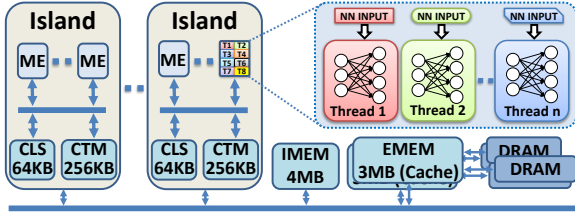it is generally dependent on the implemented data plane features [3, 8]. Furthermore, since `N3IC` leverages the same hardware in the switching chips' data plane used also for other tasks, the networking and traffic analysis functionality can usually be intertwined, e.g., in the case of programs targeting the PISA architecture the same pipeline stages may take forwarding decisions and compute BNN's neurons. This is a process that in the future may be automated too, as data plane composability technology matures [46].

BNN model generation We described in § 3 the input feature quantization and training processes for BNNs. `N3IC` applies these processes on the provided labeled dataset. For input quantization, `N3IC` takes *hints* from the programmer, who can provide the number of bits that should be used to represent each feature. For instance, the programmer may have expert knowledge about what value ranges a given feature may have. Otherwise, `N3IC` can perform an automatic assignment of features to the binary input features vector, using the range of values observed in the dataset as a guide. Once the feature quantization strategy is fixed, `N3IC` starts a model search task. During this task several models are trained, and their performance on the provided data set is tested using K-fold cross validation. Also in this case the programmer can guide the process, providing a list of models to test or limitations on the maximum model size. Our current implementation performs a simple exhaustive search over a predefined (or programmer-provided) set of models, however, this step can also be enhanced with techniques that implement more sophisticated ML architecture search solutions [38].

At the end of these two steps, `N3IC` generates a BNN model implementing an MLP architecture, described by the number of layers, number of neurons per layer, and the corresponding weights. This description is finally passed to the target-specific BNN compilers, which generate the data plane programs that implement the BNN executors. We describe these implementations next.

4.1 SoC NIC: Netronome NFP4000

The NFP4000 architecture, shown in Figure 9, comprises tens of independent processing cores, which in Netronome terminology are named micro-engines (MEs). MEs are programmed with a high-level language named *micro-C*, a C dialect. Each ME has 8 threads, which allow the system to efficiently hide memory access times, e.g., context switching

---

Algorithm 1: BNN layer processing function. Weights and inputs are in groups of `block_size`.

> **Input** : x input vector, w weights matrix, n num. of output neurons;
> **Output** : y output vector

1   $block\_size \leftarrow 32$;
2   **assert**($n \% block\_size == 0$);
3   $sign\_thr = (\mathbf{len}(x) * block\_size)/2$;
4   $y[n/block\_size] \leftarrow \{0\}$;
5   **for** $neur \leftarrow 0$ **to** $n-1$ **by** 1 **do**
6     $tmp \leftarrow 0$;
7   **for** $i \leftarrow 0$ **to** $\mathbf{len}(x)-1$ **by** 1 **do**
8       $tmp \mathrel{+}= \mathbf{popcnt}(w[neur][i] \odot x[i])$;
9   **end**
10   **if** $tmp >= sign\_thr$ **then**
11       $tmp\_out \mathrel{|}= (1 << (neur \% block\_size))$;
12   **end**
13   **if** $(neur+1) \% block\_size == 0$ **then**
14       $y[neur] \leftarrow tmp\_out$;
15       $tmp\_out \leftarrow 0$;
16   **end**
17 **end**

---

between threads as they process different packets. MEs are further organized in islands, and each island has two shared SRAM memory areas of 64KB and 256KB, called CLS and CTM, respectively. Generally, these memory areas are used to host data required for the processing of each network packet. Finally, the chip provides a memory area shared by all islands, the IMEM, of 4MB SRAM, and a memory subsystem that combines two 3MB SRAMs, used as cache, with larger DRAMs, called EMEMs. These larger memories generally host forwarding tables, access control lists, and flow counters. The BNN executor implementation has to share the MEs and memory resources with packet processing tasks, thus, it has to strike the right balance between the needs of quickly forwarding network packets and running BNN inference. For both processing tasks the main bottleneck is the memory access time. Therefore, selecting the memory area to store BNN's weights plays a major role in our design.

If the BNN is small, like in our cases, it is worth considering the fastest available on-chip memories, i.e., the CTM and CTS, with an access time of less than 100ns [29]. However, the CTM memory is usually dedicated to packet processing tasks, being the memory used by the NFP to store incoming packets and making them available to the MEs. Thus, using the CTM may impact packet processing and should be avoided. Because of this, our implementation loads the NN's weights at configuration time in the CLS memory. Then, to run the BNN, `N3IC` outputs a function that can be run within an ME's thread, and which performs Algorithm 1. This function implements the BNN executor, with input and weights packed in 32b integers (i.e., `block_size` is 32). As a consequence, multiple threads can perform BNN executions in

Algorithm 2: *popcount* implementation. $X|_y$ is the y-times concatenation of the binary number X; $Z||W$ is the concatenation of the binary numbers Z and W.

Input : n input number;
Output : c output counter

1 $B \leftarrow \lceil log_2(n+1)/8 \rceil * 8$;
2 $L \leftarrow log_2 B$;
3 $bits[L] \leftarrow \{1, 2, 4, ..., B/2\}$;
4 $masks[L] \leftarrow$
    $\{01|_{B/2}, 0011|_{B/4}, 00001111|_{B/8}, ..., 0|_{B/2}||1|_{B/2}\}$;
5 $c \leftarrow n$;
6 **for** $i \leftarrow 0$ **to** $L-1$ **by** 1 **do**
7   $\quad | \quad c \leftarrow (c \;\&\; masks[i]) + ((c >> bits[i]) \;\&\; masks[i])$;
8 **end**

parallel (Figure 9), and it is up to the programmer to decide when and how many threads to use for the BNN execution.

For example, a typical implementation would have, at boot time, each of the MEs' threads registering itself to be notified of packets reception. The NFP takes care of distributing packets to threads on a per-flow basis. This is a standard approach when programming the NFP. Thus, whenever a new packet is received, the NFP copies its content in an island's CTM, and notifies one of the island's threads to start packet processing. The notified thread can perform regular packet processing tasks, such as parsing, counters update, forwarding table lookups. The programmer can include in this context a trigger condition to start the processing of the BNN executor, by calling the function provided by N3IC. An example of triggering condition is the the reception of a predefined number of packets for a given flow.

4.2   BNN->P4->NetFPGA

P4 [6] is a domain-specific, platform-agnostic language for the programming of packet processing functions. N3IC implements a compiler that transforms BNN descriptions into BNN executors described with P4, targeting a PISA architecture. In principle, a P4-based implementation allows us to separate the N3IC's BNN executors from the underlying hardware-specific details, thus it should make the executor portable to any PISA architecture. However, as we will discuss at the end of the section, the target hardware architecture has still an important impact on the final implementation.

Compiling BNN to P4. The NNtoP4 compiler takes as input the BNN description created by the model generation step, and generates $P4_{16}$ code for a generic P4 target based on the PISA architecture. PISA is a spatial forwarding pipeline architecture, with a number of match-action units (MAUs) in series. A packet header vector (PHV), containing both the input packet and metadata information, is passed through the MAUs to perform the programmed processing tasks. Each MAU combines a table memory structure, for quick lookups using the PHV fields, with arrays of ALUs that perform operations on such fields. The code generated by NNtoP4 imple-
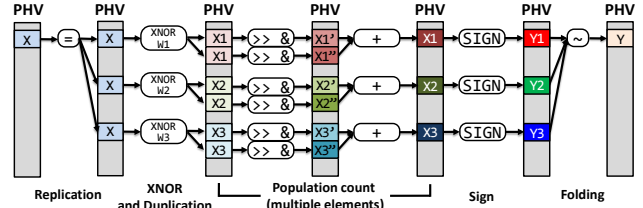

Figure 10: The logical steps required to implement a BNN using a PISA architecture.

ments a function, on top of the PISA architecture, which reads the input value from the PHV, performs the NN execution and writes back to a PHV's field the result of the computation. The NN weights are stored in the MAUs' fast memories to enable runtime reconfiguration. The generated P4 code also includes headers definition, parser, de-parser and control blocks. The code can therefore be easily extended to integrate with any other required packet processing function.

The basic operations needed to implement Algorithm 1 are (1) XNOR, (2) popcount and (3) SIGN function. Executing a XNOR and a comparison (SIGN) is readily supported by the P4 language. Unfortunately, the popcount operation is not. The main issue is that its execution time depends on the input size, which makes popcount difficult to implement in networking hardware, and therefore not supported in the PISA architecture. To overcome this issue using only current P4 primitives, we adapted the solution proposed in [4] (Item 169), as shown in Algorithm 2. The idea is to implement the popcount by combining basic integer arithmetic and logic operations in a tree structure whose depth is dependent on the input size.[3] A tree structure can be easily pipelined, with the processing of different tree's levels assigned to different pipeline's stages, thus achieving pipeline-level parallelism.

Overall, the processing includes five steps, each one mapped to a logical pipeline stage, except for the popcount which requires multiple stages, depending on the input size (cf. Figure 10). First, the NN input is replicated in as many PHV fields as the number of neurons to exploit the parallel processing on multiple packet header fields. Specifically, this corresponds to an unrolling (or partial unrolling) of the first for cycle of Algorithm 1. Second, each field, containing a copy of the NN input, is XNORed with the corresponding weight. The resulting value is further duplicated to additional fields to implement the shift, AND and sum as described in Algorithm 2. The outcome of each popcount is then compared with a threshold to implement the SIGN function, whose result is the output of each neuron. Finally, the resulting bits, stored in one PHV field for each neuron, are folded together in a single field. Depending on the NN depth, NNtoP4 replicates and concatenates the described operations as many times as the number of layers to obtain the complete MLP execution.

For hardware targets, it is worth noticing that the PHV

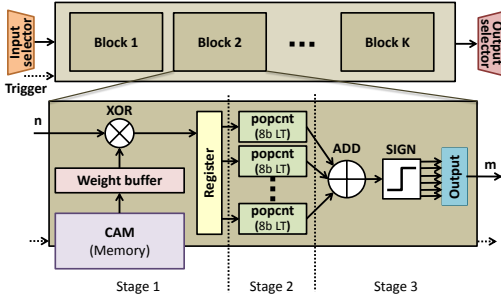[3]See [52], chapter 5, for a longer description of the algorithm.

Figure 11: Hardware design of the BNN Executor module.

size limits the number of neurons the pipeline can execute in parallel. This is due to the need to replicate the input in the PHV to enable parallelism at the MAU level.

Generating P4 code for the NetFPGA. The NetFPGA is a 4x10GbE FPGA NIC, incorporating a Xilinx Virtex-7 FPGA. We integrate `N3IC` in the reference NIC project provided with the NetFPGA-SUME code base. We used the P4->NetFPGA workflow [16] to port the generated target-independent P4 code to the NetFPGA platform. The P4->NetFPGA workflow is built upon the Xilinx P4-SDNet [54] compiler and the NetFPGA-SUME code base. It translates P4 code to Verilog, and integrates it within the NetFPGA pipeline.

The P4->NetFPGA workflow required several adaptations to the `NNtoP4` compiler, in order to meet the FPGA resources and timing constraints. First, the P4-SDNet compiler does not support `if statements` among the operations of a MAU. Thus, we replaced all the if statements required by the SIGN function using a combination of bitwise logic operations and masks. Second, MAUs use the CAM IP core from Xilinx to implement lookup tables, which restricts the maximum width size that can be used for each entry. Consequently, a maximum of 32B can be fetched from memory every time a table is called, limiting the number of neuron weights that could be loaded in parallel by each table. To overcome this issue we had to write the weights as constant values in the MAU's operations code, effectively trading the possibility to perform runtime reconfiguration with the ability to compute more neurons in parallel. Finally, P4-SDNet is capable of performing a large number of operations on a field in a single MAU. This is in contrast with ASIC targets, which are instead usually constrained to execute a single operation per MAU [45]. This allowed us to describe several steps of a BNN computation in a single MAU, thus reducing the number of MAUs required to implement the BNN computation.

5 Hardware support for BNNs

While `N3IC` can generate data plane programs that implement a BNN executor, a native support for BNNs could enable more challenging use cases. In this section, we present the implementation of a data plane's hardware primitive to run BNN, and an example of a use case that can benefit from it.

5.1 BNN inference primitive

BNN executors have been presented in the past, however, their implementations were more generally targeted to applications within devices dedicated to AI and ML workloads, e.g., cameras. Instead, our target is to design a BNN executor integrated within the data plane of a NIC. This changes the implementation constraints. Most notably, our executor targets smaller models, and it is designed to fetch input data from the internal data plane data buses. We target the NetFPGA prototyping platform, and design our BNN executor in HDL.

Figure 11 shows the architecture of our BNN executor. The module is composed of multiple blocks. Each of them performs the computation of a single NN layer, and can be parametrized providing the sizes n and m for the input and output vectors, respectively. Together, the blocks build a BNN Executor for specific BNN architectures. For instance, three of these blocks are required to build a 3 layers MLP. The NN layer weights are stored in the FPGA on-chip memories, i.e., Block RAM (BRAM). The BRAMs are organized as tables with a number of rows dependent on the number of neurons, and with a width of 256b. Each row can be read in 2 clock cycles and, depending on the size n of the input vector, can store one or multiple weights, e.g., 1x256b or 16x32b. The BRAMs are shared by all the blocks of a BNN module.

A single block is a pipeline of three stages. The first reads the weights from the BRAM and performs the XNOR with the input. The second performs the first step of the popcount. Here, we create Lookup-Tables (LTs) of 256 entries each, in order to associate one 8b integer (address) to the corresponding population count value. Each block has $n/8$ of these LTs. As a consequence, for a 256b input we create 32 LTs that operate in parallel. In the last stage, the LTs outputs are summed together, the sign function is applied on the final sum and its result is stored in one of the m bits of the output register. If multiple weights are placed in a single BRAM's row, the module performs the execution of several neurons in parallel.

5.2 Enabling more challenging use cases

The BNN inference primitive can enable more challenging applications that have very low processing latency requirements. To highlight this, we look at a recently presented network tomography solution: SIMON [14]. SIMON periodically sends probe packets to measure network path delays, and then it uses the collected delay measurements to infer congestion points and the size of the related queues. The analysis of probe delays is performed offline with neural networks (MLPs). The high processing latency only enables post-mortem analysis. Therefore, in its current implementation, SIMON cannot be used to create a measurement and control loop, i.e., for path selection. The probe periodicity defines the processing latency constraint and it depends on the fastest link speed [14]. For instance, probes have to be sent every 250µs and 100µs for 40Gb/s and 100Gb/s links, respectively. As a consequence, to work at modern datacenters' link speeds and in real-time, the
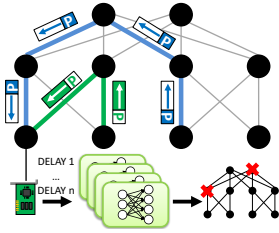
Figure 12: N3IC enables the real-time implementation of SIMON [14], using BNNs in the NIC to identify congested queue from probes' one-way delays. This can be used to implement new traffic steering policies in the data plane.
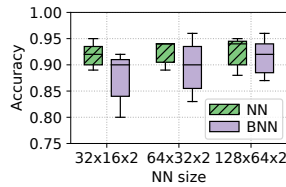
Figure 13: Box plot of the accuracies for the predicted queues in the network tomography use case. BNNs makes our approach practical while trading just a tiny amount of accuracy with respect to non-binarized NNs.

execution latency has to be lower than few tens of µs.

We tested the use case simulating a CLOS-like Fat Tree datacenter network with ns3 [50], using different link speeds and traffic workloads. Following the methodology suggested by [14], we split the problem of inferring queue sizes in multiple sub-problems, each targeting a subset of the queues. This allows us to run smaller MLPs on each of the NICs. Unlike SIMON, our approach does not infer the actual size of a queue, but it only infers which queues are bigger than given thresholds levels. This information is usually sufficient for the control plane to take a flow-steering decision (See Figure 12).

We implement SIMON with N3IC, providing as input features 19 probes' one-way delays per BNN. A NIC can run multiple BNNs, since each of them infers the congestion status of a specific queue. We show the accuracy of prediction for each of the network queues in Figure 13, comparing the BNNs accuracy to that of non-binarized neural networks. For a BNN with three layers and 128, 64, 2 neurons per layer, across all the queues of the simulated network, we achieve a median accuracy in predicting a congested queue above 92%, which is comparable with the non-binarized neural network accuracy. As we will see in § 6, the introduced BNN hardware primitive will enable running these BNNs within the processing latency required for links faster than 400Gb/s.

6 System-level Evaluation

In this section, we present the experimental evaluation of N3IC's BNN executors. We report and discuss the end-to-end performance of the use cases presented in § 3, and of the the network tomography use case from § 5. Furthermore, we report results for micro-benchmarks and resource requirements.

Testbed. Unless stated otherwise, the system-under-test (SuT) uses a machine equipped with an Intel Haswell E5-1630 v3 CPU and either a Netronome Agilio CX, with an NFP4000 processor, or a NetFPGA-SUME[4]. The Haswell is clocked at

---

[4]The Haswell CPU was produced with a 22nm factory process, i.e., a technology comparable to the NFP4000 (22nm) and NetFPGA Virtex7 (28nm).

3.7GHz, the NFP at 800MHz, and the NetFPGA at 200MHz for both the N3IC-P4 and N3IC-FPGA (i.e., using the hardware primitive) implementations. The host system runs Linux, kernel v.4.18.15. The SuT is connected back-to-back to a second machine that hosts the traffic generators and receivers. For stress tests, we use a 40Gb/s capable DPDK packet generator[5], and we use HTTP clients and ngnix as receiver, both hosted on the second machine. We always measure that the SuT is the performance bottleneck, ensuring that the setup achieves line-rate when removing the SuT from the loop.

Comparison term. We compared our prototypes with a traffic analysis system (bnn-exec) that performs feature extraction on the NIC and the analysis task in software, using binary neural networks like those employed by N3IC. bnn-exec is available at [43]. We wrote bnn-exec in C, and optimized it for the Haswell CPU, with some parts in assembler to take full advantage of the CPU's architecture features, such as AVX2 instructions. bnn-exec is faster than any other software BNN executor we tested, and performs the analysis task with performance comparable to that of optimized libraries for DTs and RFs [17]. We setup bnn-exec to read flows statistics/data from the Netronome NIC and ran bnn-exec only with the Netronome NIC since its driver is more mature than the NetFPGA's: it can better handle fast communication between the NIC and the host system. When performing analysis with bnn-exec we took into account (1) the time to read one or more flow statistics; (2) the time to run the BNN itself; and (3) the time to write back the result on the NIC. This allows us to perform a fair comparison against N3IC.

Feature extraction. Our end-to-end system need feature extraction to be implemented in the NIC's data plane.In fact, the quality of the inference tasks performed by the downstream ML model strictly depends on the quality of the extracted features. In some use cases, feature extraction may be simpler than in others. For instance, in the IoT use case the outcome of the inference task assigns flows to QoS classes. While a mis-classification is undesirable, its impact on the infrastructure is usually limited, and one may give priority to efficiency of implementation vs accuracy. Instead, in security use cases there may be a stricter need to ensure that feature extraction is robust, e.g., to protect against adversarial attacks [37].

For the tests in this section, we use two different state-of-the-art feature extraction strategies. In stress tests, we use a simpler approach that allows us to evaluate the N3IC implementations, ensuring that N3IC is the actual system bottleneck. In this case, the NIC stores the per-flow features in a hashtable, using the flow's 5-tuple as lookup key. When a packet is received, the corresponding flow's features are retrieved from the hashtable and updated. If the lookup produces a miss, the packet is considered as belonging to a new flow. Entries are removed from the hashtable lazily, if no packets for the corresponding flow are received in a given time window. In
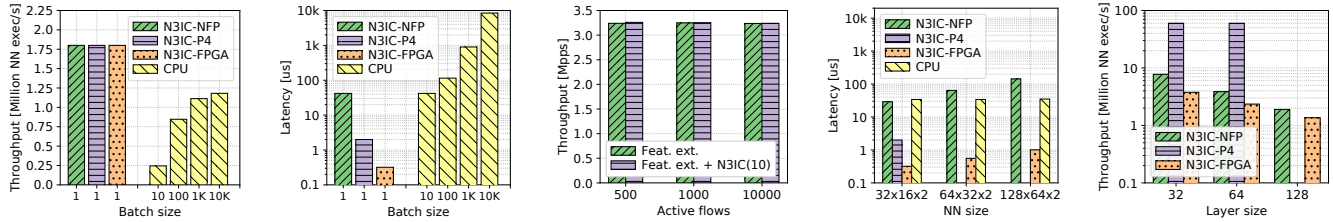
---

[5]https://git.dpdk.org/apps/pktgen-dpdk/

Figure 14: For IoT and Security use cases, `N3IC` is 1.5x-7x faster than `bnn-exec`, while forwarding packets at 40Gb/s.

Figure 15: For IoT and Security use cases, `N3IC` implementations can provide at least 10-100x lower latency than `bnn-exec`.

Figure 16: `N3IC`-NFP throughput for the IoT use case when analyzing 500, 1k, and 10k parallel TCP flows.

Figure 17: `N3IC`-FPGA can support the network tomography use case even in fast 400Gb/s networks with probes sent every 25µs.

Figure 18: Maximum throughput in number of BNNs execution per second for `N3IC` BNN executors.

this approach, only the TCP's connection establishment is tracked, and no further connection tracking is performed.

In a second approach, we use a more complex solution performing full TCP-connection tracking. A connection tracking automaton validates that a received packet belongs to the 5-tuple flow (e.g., checking sequence numbers), before performing the features update as in the simpler approach. We used the TCP-connection tracking implementation of Flowblaze [36] that also allows us to change the behavior for sequence number checking, e.g., using either *window shifting* or *window advancing* solutions.

In both cases, we only track flow-level features. Collecting the 3 host-level features used in the Security use case requires more complex operations, which we did not implement since the impact of such features on the BNN classification accuracy is negligible (Cf. Appendix for a detailed report). Flow-level features can be: directly extracted from the packet headers (e.g. protocol number), computed by accumulating values extracted from packet headers (e.g, total transferred bytes) or derived from the calculation of flow level metrics (e.g. packet interarrival, mean flow size). In the latter case difference based metrics (e.g. flow duration) are computed for each packet in the flow, while mean based metrics are only partially computed (i.e. total and number of values are stored separately) per packet and then finalized (i.e. total/number) each time the feature is fed to the NN. Additional per flow values are stored in order to compute the flow level metrics (Table 4 in Appendix). The computation of the per-flow statistics is a memory-bound operation so the extra overhead due to the metric computation is negligible respect to the cost of accessing the flow tables. We implement the feature extraction strategies both in the Netronome NFP and in the NetFPGA, for which we report its resources consumption in Table 1. We refer to the two Feature Extraction (FE) strategies as *simple* FE and *advanced* FE, respectively.

6.1   End-to-end performance tests

In all the end-to-end tests, we measure the analysis throughput and latency, *while the system-under-test forwards network*

traffic at 40Gb/s within the NIC (NFP4000 or NetFPGA).

Traffic analysis use cases We perform two different tests to measure the end-to-end `N3IC` performance with the use cases from Section 3. First, we run a stress test generating a large number of small packets with the DPDK packet generator, then we perform a performance test with real TCP flows generated by HTTP clients and nginx. For the stress test, the provided traffic contains 1.8M flows per second.[6] This is a challenging load for a single server, being more common in ToR switches handling traffic for high throughput user-facing services [23]. If `N3IC` can meet this performance goal, it is likely to be capable of handling a large range of ordinary use cases. For the TCP tests, we vary the number of flows between 500 and 10k, and always generate 40Gb/s of traffic. *Baseline:* We measured the NIC performance when only collecting flow statistics with the simpler approach introduced earlier. The Netronome provides its 40Gb/s line rate only with packets of size 256B (18.1Mpps) or bigger. This is achieved using 90 out of the 480 available threads, and it is in line with the device's expected performance for such class of applications. In fact, the NFP can efficiently hide hash-table lookup latencies by distributing the processing on multiple threads, while consistently assigning flows to different threads. This avoids expensive locking of the hash-table, since different parallel executors do not access the same entry. The NetFPGA, instead, is capable of forwarding 40Gb/s with minimum size (64B) packets while collecting flow statistics, in any case. *Stress Tests:* We use the smaller BNN models reported in § 3 to test `N3IC` performance, since they achieve comparable accuracy with the larger DTs and RFs models. We summarized the throughput results in Figure 14. `N3IC` implementations can all achieve the offered throughput of 1.81M flow analysis/s. Instead, even if using larger batch sizes, `bnn-exec` is unable to cope with such load, when running on a single CPU core. `bnn-exec` maximum throughput is 1.18M analyzed flows/s, when using very large batches of 10K flows. More interestingly, Figure 15 shows that `N3IC` implementations provide also a low processing latency, with a 95-th percentile of 42µs

---

[6]That is, an average of 10 packets per flow at 40Gb/s@256B.

for `N3IC`-NFP, and only 2µs and 0.5µs for `N3IC`-P4 and `N3IC`-FPGA, respectively. In comparison, for `bnn-exec` to achieve a throughput above the 1M flows/s, the processing latency is 1ms and 8ms with batch sizes 1K and 10K, respectively.

TCP Test: we run an additional experiment, using the IoT application, to check the functionality of `N3IC` with flows generated by HTTP clients and *nginx*, when using the second feature extraction strategy with full TCP tracking. The HTTP clients generate 40Gb/s distributed among 500, 1k, and 10k parallel flows. Since TCP flows have larger average packet size (close to the maximum of 1.5KB), this corresponds to about 3.2Mpps at 40Gb/s line rate. We further instrument `N3IC` to perform inference on a flow after every (10, 100, 1000) received flow's packets. This corresponds to up to over 320k ML inferences per second. Figure 16 shows that `N3IC` can forward all the received packets, while collecting statistics and performing ML inference (we show only `N3IC`-NFP, and for inferences every 10 flow's packets, since results for the other experiments and for `N3IC`-FPGA are similar). An interesting observation is that the NFP's throughput does not change when adding `N3IC` inference load. This happens since feature extraction requires memory lookups, whereas BNN inference requires mostly processing power from the NFP's MEs, thus the two workloads can be efficiently co-located.

Network Tomography When testing the network tomography use case, the NIC stores the one-way-delay value for the received network probes, before passing them to the analysis engine, i.e., either `N3IC` or `bnn-exec`. Here, processing latency is the critical performance indicator. Figure 17 shows that `bnn-exec` provides a processing latency of about 40µs, which is within the budget of 100µs.[7] However, upcoming network links of 400Gb/s could not be supported, since they would lower the periodicity of the probes to 25µs. `N3IC` processing latency for SIMON's BNNs with 128, 64, 2 neurons is 170µs for `N3IC`-NFP and below 2µs for `N3IC`-FPGA. As we further clarify next, `N3IC`-P4 cannot scale to run larger BNNs, and can only run the smaller 32, 16, 2 neurons networks with about 2µs of delay, at the cost of reduced accuracy. For upcoming 400Gb/s network speeds, the BNN hardware primitive enables running more accurate BNN models, while being within the processing latency requirement.

6.2 Scalability tests

We now evaluate the processing throughput and latency when varying the size of the BNN. We performed this evaluation fully loading `N3IC`, and by executing a single BNN layer with 256 binary inputs. We varied the number of neurons to be 32, 64, and 128.[8] Figure 18 shows that the throughput decreases linearly with the layer's size for `N3IC`-NFP and `N3IC`-FPGA. Latency, instead, increases linearly (not shown). This is ex-

---

[7]In this case high-throughput is not required, so we use a batch size of 1.

[8]The layer is fully-connected, therefore its size is the number of input times the number of neurons: a layer with 128 neurons has 4KB of weights, i.e., about 4x the size of the NN used for the traffic analysis use cases.

Design	LUT		BRAM	
	#	% tot	#	% tot
Reference NIC (RN)	49.4k	11.4%	194	13.2%
RN + simple Feature Extraction (FE)	50.0k	11.56%	258	17.6%
RN + simple FE + `N3IC`-FGPA	52.6k	12.16%	275	18.8%
RN + simple FE + `N3IC`-P4	145.1k	33.56%	582	39.6%
RN + advanced FE	92.0k	21.56%	458	32.6%
RN + advanced FE + `N3IC`-FPGA	95.0k	22.86%	475	33.8%

Table 1: NetFPGA resources usage. `N3IC`-FPGA requires little additional resources. `N3IC`-P4 uses a large amount of NIC resources due to the PISA computation model constraints.

pected given the design presented in § 4. In comparison, `N3IC`-P4 throughput results are much higher for a layer with 32 and 64 neurons. Unfortunately, results for 128 neurons are missing, since `N3IC`-P4 could not scale to handle such layers. We provide more insight on this in the next subsection.

6.3 System resources usage

We quantify the resources needed by `N3IC`. Compared to state-of-the-art systems like `bnn-exec`, `N3IC` does NOT use any CPU cores and keeps the PCIe bus free. It does however use additional resources on the NIC. We evaluate this referring to the BNNs used in the traffic analysis use cases.

In the NFP case, `N3IC` has to store the NN's weights in the NFP4000's memory system. The NNs used with the traffic analysis use cases require 1.5% of the CLS memory, and 480 threads to face the offered load, instead of the 90 required to achieve line-rate throughput when the NIC is only collecting flow statistics. Here, it should be noted that it is possible to use less threads, if a performance drop in NN inference throughput is acceptable. For instance, using only 120 threads, i.e., 30 additional threads compared to the baseline, reduces the throughput of flows analyzed per second by 10x. This still provides the ability to analyze over 100k flows per second, which is sufficient for many workloads.

In the NetFPGA cases, we measured the hardware resources required to synthesize `N3IC` on the Virtex7 FPGA, and compare them to the standard NetFPGA reference NIC design's resources, including the resources required to implement the feature extraction logic. Table 1 summarizes the results. `N3IC`-FPGA requires only an additional 0.6% and 1.2% of the FPGA's LUTs and BRAMs, respectively. The resource consumption is so small since we included in the data plane a single BNN executor module, which was dimensioned to achieve the analysis throughput measured in the tests reported in this section. Instead, the `N3IC`-P4 implementation requires a relatively large amount of resources, with an additional 22% for both LUTs and BRAMs. For comparison, the implementation of DTs with depth 5 in the data plane reported in [55] requires 27% and 40% of LUTs and BRAMs, respectively. This is the case because the P4 implementation embeds the BNN executor within a PISA-like pipeline targeted by the P4->NetFPGA toolchain. That is, the computations of the BNN are unrolled to be distributed on multiple PISA's

match-action stages. While this has the effect of completely pipelining the BNN execution, it also requires using a large amount of FPGA resources. That is, like it is the case for other P4 programs using the P4->NetFPGA toolchain, with `N3IC`-P4 the successful compilation and synthesis of the P4 program guarantees the NIC's line rate during execution. Therefore, `N3IC`-P4 can run a BNN inference for each received packet and still match packet forwarding line rate (cf. Figure 18). For this reason, it should also be noted that in `N3IC`-P4 most of the resources can be reused to implement also regular packet forwarding, since the pipeline stages required by `N3IC` can host forwarding rules coming from other processing tasks.

7   Discussion

What are the limitations? Since `N3IC` BNNs run in the data plane, only features that can be computed/extracted within the data plane can be used as input. This limits the applicability of `N3IC` to devices that offer such functionality. For instance, porting `N3IC` to a switch's data plane may be limited by the availability of input features. For similar reasons, more complex models that require application-level data, e.g., payload of packets and with KBs of input size, are not well handled by `N3IC`. For these kind of analysis tasks, more relevant solutions may be previous work such as Brainwave [24] and Taurus [47], or some recently presented NICs that combine specialized executors for ML models, e.g., NVIDIA EGX A100 [32] and Xilinx Alveo SN1000 [53]. In fact, although these executors are not well suited for the low latency analysis tasks addressed by `N3IC` (cf. § 2), they are especially designed to perform complex algorithms on larger data, with processing latency in the ms.

Is it all about scalability and performance? While `N3IC` improves the performance of existing traffic analysis systems, we believe the ability to perform flow-level traffic analysis entirely in the NIC can provide a tool to rethink system architectures. For instance, the ability to track the queue status of network switches in near real time (§ 5.2) would make it practical the implementation of load-aware data center load balancing schemes that take decision from the end host [18], or it could enable new congestion control algorithms.

8   Related Work

Traffic analysis with machine learning is performed by systems in many operational settings [39], e.g., for traffic classification [2, 5, 14, 26, 40] and security [3, 10, 19, 25]. Some solutions scale traffic analysis performance using NICs [12, 29, 48] that have the ability to perform feature extraction (e.g., flow statistic collection [1, 3, 28]). Unlike these solutions, `N3IC` enables also the execution of machine learning-based analysis within the NIC's data plane. Previous work presented a similar idea when targeting switches [8, 55], and [8] covers also the issue of selecting the subset of features that can be efficiently collected within the data plane. In `N3IC` we leverage the flexibility of a NIC's data plane, designed to process

significantly less traffic than a switch, to relax this issue.

The idea of using binary neural networks within the network data plane was presented in some early works [41, 42]. [42] presents a conceptual design for RMT [7] switches. [41] targets end-host ML applications, in which the NIC works as a co-processor for Convolutional Neural Networks for image classification that runs on the host. We build on similar insights and extend those early ideas in many ways. First, we show the suitability of BNNs for traffic analysis use cases, comparing them with state-of-the-art ML techniques. Then, we present an end-to-end system design that builds BNN executors for different NIC architectures, starting from a labeled dataset. Finally, we present a complete evaluation of BNN executors on two NICs, propose a dedicated hardware-native implementation, and include an end-to-end evaluation of three networking use cases, with related trade-offs.

Finally, while not directly related to `N3IC`, recent work on the security of network applications that use machine learning is likely to influence developments in this area [22, 31].

9   Conclusion

We addressed the problem of improving throughput, latency, and efficiency of packet- and flow-level network traffic analysis, usually performed by software middleboxes and network functions. We first show that binary neural networks can replace widely-adopted decision trees and random forests, on the tested network traffic analysis tasks. Then, we make the case for implementing them in the data plane of commodity programmable NICs. We design and implement an end-to-end system composed of a binary neural network model generation module, and a compiler that generates data plane programs to execute the binary neural network model in the data plane of commodity programmable NICs (i.e., Netronome SmartNICs and P4-enabled NICs). Moreover, we also design and prototype a new hardware primitive that allows a NIC to perform BNN model execution directly. We evaluated our approach using two different NICs, Netronome NFP4000 and NetFPGA, and for a set of use cases representing a large variety of current traffic analysis applications, including traffic classification, anomaly detection and network tomography. Our results show that our system can accurately perform analyses for millions of flows per second, with low latency, while processing packets at NICs' line rates.

References

[1] Accolade Technology. ANIC Host CPU Offload Features Overview, [Online; accessed 04-March-2021]. https://accoladetechnology.com/whitepapers/ANIC-Features-Overview.pdf.

[2] Giuseppe Aceto, Domenico Ciuonzo, Antonio Montieri, and Antonio Pescapé. Mobile encrypted traffic classification using deep learning: Experimental evaluation, lessons learned, and challenges. *IEEE Transactions on Network and Service Management*, 16(2):445–458, 2019.

[3] Diogo Barradas, Nuno Santos, Lui Rodrigues, Salvatore Signorello, Fernando M.V. Ramos, and Andre Madeira. Flowlens: Enabling efficient flow classification for ml-based network security applications. In *Network and Distributed Systems Security (NDSS)*. USENIX, 2021.

[4] M. Beeler, R.W. Gosper, and R. Schroeppel. Hakmem AI Memo No. 239. In *MIT Artificial Intelligence Laboratory, Cambridge, US*, 1972.

[5] Laurent Bernaille, Renata Teixeira, Ismael Akodkenou, Augustin Soule, and Kave Salamatian. Traffic classification on the fly. *SIGCOMM Comput. Commun. Rev.*, 36(2):23–26, April 2006.

[6] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming Protocol-independent Packet Processors. In *Computer Communication Review, Volume: 44, Issue: 3*. ACM, 2014.

[7] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2013.

[8] Coralie Busse-Grawitz, Roland Meier, Alexander Dietmüller, Tobias Bühler, and Laurent Vanbever. pforest: In-network inference with random forests. *arXiv preprint arXiv:1909.05680*, 2019.

[9] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshuao Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. In *Computing Research Repository, Volume: abs/1602.02830*, 2016.

[10] Luca Deri. Using ndpi for monitoring and security, 2021. https://fosdem.org/2021/schedule/event/nemondpi/.

[11] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 523–535, Santa Clara, CA, March 2016. USENIX Association.

[12] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mark Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish K. Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *Networked Systems Design and Implementation (NSDI)*. USENIX, 2018.

[13] Gartner. Market guide for network detection and response, 2020. https://www.gartner.com/doc/reprints?id=1-25DOQJMT&ct=210304&st=sb.

[14] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. SIMON: A Simple and Scalable Method for Sensing, Inference and Measurement in Data Center Networks. In *Networked Systems Design and Implementation (NSDI)*. USENIX, 2019.

[15] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In *Neural Information Processing Systems (NIPS)*. Curran Associates Inc., 2016.

[16] Stephen Ibanez, Gordon Brebner, Nick McKeown, and Noa Zilberman. The p4->netfpga workflow for line-rate packet processing. In *Field-Programmable Gate Arrays (FPGA)*. ACM, 2019.

[17] intel. Daal, 2019. https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onedal.html.

[18] Naga Katta, Aditi Ghag, Mukesh Hira, Isaac Keslassy, Aran Bergman, Changhoon Kim, and Jennifer Rexford. Clove: Congestion-Aware Load Balancing at the Virtual Edge. In *Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*. ACM, 2017.

[19] Hongda Li, Hongxin Hu, Guofei Gu, Gail-Joon Ahn, and Fuqiang Zhang. VNIDS: Towards Elastic Security

with Safe and Efficient Virtualization of Network Intrusion Detection Systems. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 17–34, New York, NY, USA, 2018. Association for Computing Machinery.

[20] Xiaofan Lin, Cong Zhao, and Wei Pan. Towards accurate binary convolutional neural network. In *Neural Information Processing Systems (NIPS)*. Curran Associates, Inc., 2017.

[21] Francis Matus. Distributed services architecture. In *2020 IEEE Hot Chips 32 Symposium (HCS)*, pages 1–17. IEEE Computer Society, 2020.

[22] Roland Meier, Thomas Holterbach, Stephan Keck, Matthias Stähli, Vincent Lenders, Ankit Singla, and Laurent Vanbever. (Self) Driving Under the Influence: Intoxicating Adversarial Network Inputs. In *Hot Topics in Networks (HotNets)*. ACM, 2019.

[23] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. SilkRoad: making stateful layer-4 load balancing fast and cheap using switching ASICs. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2017.

[24] Microsoft. Microsoft unveils project brainwave for real-time ai, 2017. https://www.microsoft.com/en-us/research/blog/microsoft-unveils-project-brainwave/.

[25] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. Kitsune: An ensemble of autoencoders for online network intrusion detection. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.

[26] Andrew W. Moore and Denis Zuev. Internet Traffic Classification Using Bayesian Analysis Techniques. In *Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*. ACM, 2005.

[27] Nour Moustafa and Jill Slay. UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set). In *Military Communications and Information Systems Conference (MilCIS)*. IEEE, 2015.

[28] Napatech. SmartNICs features overview, [Online; accessed 04-March-2021]. https://www.napatech.com/support/resources/data-sheets/napatech-smartnic-feature-overview/.

[29] Netronome. Netronome AgilioTM CX 2x40GbE intelligent server adapter, 2018. https://www.netronome.com/media/redactor_files/PB_Agilio_CX_2x40GbE.pdf.

[30] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. Understanding PCIe Performance for End Host Networking. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2018.

[31] Carlos Novo and Ricardo Morla. Flow-Based Detection and Proxy-Based Evasion of Encrypted Malware C2 Traffic. In *ACM Workshop on Artificial Intelligence and Security*, AISec'20, New York, NY, USA, 2020. Association for Computing Machinery.

[32] NVIDIA. Nvidia egx a100, 2018. https://www.nvidia.com/en-us/data-center/products/egx-converged-accelerator/.

[33] OVH. Managing a ddos attack, 2021. https://www.ovh.com/world/anti-ddos/managing-ddos-attacks.xml.

[34] OVH. What is anti-ddos protection?, 2021. https://www.ovh.com/world/anti-ddos/anti-ddos-principle.xml.

[35] Vern Paxon. The Zeek Network Security Monitor, [Online; accessed 04-Feb-2020]. https://www.zeek.org/.

[36] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, Felipe Huici, and Giuseppe Bianchi. FlowBlaze: stateful packet processing in hardware. In *Networked Systems Design and Implementation (NSDI)*. USENIX, 2019.

[37] Zhiyun Qian and Z. Morley Mao. Off-path tcp sequence number inference attack - how firewall middleboxes reduce security. In *2012 IEEE Symposium on Security and Privacy*, pages 347–361, 2012.

[38] Pengzhen Ren, Yun Xiao, Xiaojun Chang, Po-yao Huang, Zhihui Li, Xiaojiang Chen, and Xin Wang. A comprehensive survey of neural architecture search: Challenges and solutions. 54(4), 2021.

[39] Paulo Angelo Alves Resende and André Costa Drummond. A survey of random forest based methods for intrusion detection systems. *ACM Comput. Surv.*, 51(3), May 2018.

[40] Said Jawad Saidi, Anna Maria Mandalari, Roman Kolcun, Hamed Haddadi, Daniel J Dubois, David Choffnes, Georgios Smaragdakis, and Anja Feldmann. A haystack full of needles: Scalable detection of iot devices in the wild. In *Proceedings of the ACM Internet Measurement Conference*, pages 87–100, 2020.

[41] Davide Sanvito, Giuseppe Siracusano, and Roberto Bifulco. Can the network be the AI accelerator? In *In-Network Computing (NetCompute)*. ACM, 2018.

[42] Giuseppe Siracusano and Roberto Bifulco. In-network neural networks. In *Computing Research Repository, Volume: abs/1801.05731*, 2018.

[43] Giuseppe Siracusano, Salvator Galea, Davide Sanvito, Mohammad Malekzadeh, Gianni Antichi, Paolo Costa, Hamed Haddadi, and Roberto Bifulco. N3ic github repository, 2022. https://github.com/nec-research/n3ic-nsdi22.

[44] Arunan Sivanathan, Hassan Habibi Gharakheili, Franco Loi, Adam Radford, Chamith Wijenayake, Arun Vishwanath, and Vijay Sivaraman. Classifying iot devices in smart environments using network traffic characteristics. *IEEE Transactions on Mobile Computing*, 18(8):1745–1759, 2018.

[45] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: high-level programming for line-rate switches. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2016.

[46] Hardik Soni, Myriana Rifai, Praveen Kumar, Ryan Doenges, and Nate Foster. Composing dataplane programs with μp4. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 329–343, 2020.

[47] Tushar Swamy, Alexander Rucker, Muhammad Shahbaz, Neeraja Yadwadkar, Yaqi Zhang, and Kunle Olukotun. Taurus: An Intelligent Data Plane. In *P4 Workshop*, 2019.

[48] Mellanox Technologies. BlueField Smart-NIC, 2019. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_BlueField_Smart_NIC.pdf.

[49] Mellanox Technologies. TILEncore-Gx72, 2019. https://www.mellanox.com/page/products_dyn?product_family=231&mtag=tilencore_gx72_adapter_mtag.

[50] The University of Washington NS-3 Consortium. NS3 official website, [Online; accessed 10-Jan-2020]. https://www.nsnam.org/.

[51] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. In *Field-Programmable Gate Arrays (FPGA)*. ACM, 2017.

[52] Henry S. Warren. *Hacker's Delight*. Addison-Wesley Professional, 2nd edition, 2012.

[53] XILINX. Xilinx sn1000, 2018. https://www.xilinx.com/applications/data-center/network-acceleration/alveo-sn1000.html.

[54] Xilinx. SDNet compiler, 2019. https://www.xilinx.com/sdnet.

[55] Zhaoqi Xiong and Noa Zilberman. Do switches dream of machine learning? toward in-network classification. In *Proceedings of the 18th ACM workshop on hot topics in networks*, pages 25–33, 2019.

A Appendix

We provide additional details and test results about the tested machine learning (ML) models, and about the implementation of the BNN executors.

A.1 Input Features

Table 2 reports the set of features used by the IoT Traffic Classification use case to perform the classification, while Table 3 reports the ones used by the Security Anomaly Detection use case. It should be noted that all the features used by the former use case are also used by the latter. However, some of the features shared by the two use cases differ in the number of bits used for their binary encoding. For example, feature *dur* in the IoT use case requires twice the number of bits with respect to the Security use case.

For the Security use case, we tested the classification performance of the ML models with and without the host-based features. In fact, these features complicate significantly the feature extraction process on the NIC. As we will see in the next subsections, this impacts significantly the classification performance of Decision Trees, while it has minimal impact on BNNs. We speculate that this is the case since not only BNNs perform classification using all the available features, but they also naturally build intermediate features (i.e., feature engineering) in their hidden layers; whereas DTs and RFs use only a subset of the provided features. This observation suggests that there maybe more advantages in using BNNs, beyond those reported in the paper. E.g., BNNs may enable to perform inference using a set of features that are cheaper to collect. However, we leave more investigation into this for future studies, and therefore we only report that this is indeed the case for the Security use case.

Features number vs Memory requirements. Another aspect we did not discuss in the paper is the memory requirement associated with the features. This is usually a bigger issues in switching devices that deal with larger amounts of traffic, such as network switches and routers, while it is not a hard constraint in NICs that are provided with larger (per-flow) memories. In the use cases analyzed in the paper, we use a 256b feature vector, i.e., each flow entry has a memory occupation of 45B (13B for the flowkey, and 32B for the features). That is, a features table for e.g., 10K active flows needs less than 0.5MB of (SRAM) memory.

Feature extraction additional counters. Table 4 lists additional counters that are needed for the feature extraction. Indeed, in order to calculate duration and average input features, five per flow counters have to be stored. Timestamps of the flow start and the last packet sent by the source/destination are used to calculate: the duration of the flow (dur Table 2, 3), the average load (sload, dload), the interarrival times (sinpkt and dinpkt) and TCP connection setup time (ackdat, synack).

Feature	Description	Bin. enc. length
dur	record total duration	16
proto	transaction protocol	8
sbytes	src -> dst transaction bytes	24
bytes	dst -> src transaction bytes	24
sttl	src -> dst TTL value	8
dttl	dst -> src TTL value	8
sload	source bits per second	24
dload	destination bits per second	24
spkts	src -> dst packet count	16
dpkts	dst -> src packet count	16
smean	Mean of the flow packet size tx by the src	16
dmean	Mean of the flow packet size tx by the dst	16
sinpkt	source interpacket arrival time	16
dinpkt	destination interpacket arrival time	16
tcprtt	TCP connection setup round-trip time the sum, of 'synack' and 'ackdat'.	8
synack	TCP connection setup time, the time between, the SYN and the SYN_ACK packets	8
ackdat	TCP connection setup time, the time between the SYN_ACK and the ACK packets.	8

Table 2: IoT Traffic Classification input features

While the source/destination total packet counters are used only to calculate the mean flow size.

A.2 Machine Learning Models

A.2.1 Additional evaluation metrics

This section provides supplementary evaluation results to complement the F1-score and False Positive Rate (FPR) metrics presented in Section 3 of the paper. TP, TN, FP and FN indicate the True Positives, True Negatives, False Positives, and False Negatives, respectively. We report here the following metrics:

- Accuracy: computed as $(TP + TN)/(TP + TN + FP + FN)$, it quantifies the percentage of correct predictions.
- Precision (P): computed as $TP/(TP + FP)$, it quantifies the quota of positive class predictions that actually belong to the positive class.
- Recall (R) or True Positive Rate (TPR): computed as $TP/(TP + FN)$, it quantifies the quota of positive samples that are correctly predicted as positive.
- F1-score: computed as $2TP/(2TP + FP + FN)$, it is the harmonic mean of Precision and Recall.
- False Positive Rate (FPR): computed as $FP/(FP + TN)$, it quantifies the quota of negative samples that are wrongly predicted as positive.
- False Negative Rate (FNR): computed as $FN/(FN + TP)$, it quantifies the quota of positive samples that are wrongly predicted as negative.
- ROC-AUC: the Receiver Operating Characteristic (ROC) curve captures the TPR-FPR tradeoff at different classification thresholds. ROC-AUC is the area under the ROC curve and provides an aggregate measure to quantify the performance of a classification model across all the classification thresholds.

Feature	Description	Bin. enc. length
dur	record total duration	8
proto	transaction protocol	8
sbytes	src -> dst transaction bytes	16
bytes	dst -> src transaction bytes	16
sttl	src -> dst TTL value	8
dttl	dst -> src TTL value	8
sload	source bits per second	24
dload	destination bits per second	24
spkts	src -> dst packet count	16
dpkts	dst -> src packet count	16
smean	Mean of the flow packet size tx by the src	16
dmean	Mean of the flow packet size tx by the dst	16
sinpkt	source interpacket arrival time	16
dinpkt	destination interpacket arrival time	16
tcprtt	TCP connection setup round-trip time, the sum of 'synack' and 'ackdat'.	8
synack	TCP connection setup time, the time between the SYN and the SYN_ACK packets	8
ackdat	TCP connection setup time, the time between the SYN_ACK and the ACK packets	8
Host-based features		
ct_src_ltm	No. of connections of the same dst address in 100 connections according to the last time	8
ct_dst_ltm	No. of connections of the same src address in 100 connections according to the last time	8
ct_ds_src_ltm	No of connections of the same src/dst address in 100 connections according to the last time	8

Table 3: Security Anomaly Detection input features

Counter	Description
flow start	flow start timestamp
dst pkt count	Total number of packets sent by dst
src pkt count	Total number of packets sent by src
dts last pkt ts	Timestamp of the last pkt sent by dst
src last pkt ts	Timestamp of the last pkt sent by src

Table 4: Feature extraction additional counters

Here, we notice that the False Negative Rate (FNR) is not reported in the results, since it is computed as $FNR = 1 - Recall$, and we already report Recall for all the cases.

For each metric we report the average and standard deviation resulting from a 5-fold cross-validation. In the IoT case we are dealing with a 10-classes classification problem, thus, we used a one-vs-rest strategy to evaluate the False Positive and True Positive Rates. Following the description of Section 3, we focused on 3 representative configurations for each type of model. Specifically, for the Decision Tree (DT) and Random Forest (RF) models we considered tree depths values of 3, 6, and 9, and always 5 trees for the RF. The BNN models use a Multi-layer Perceptron architecture, with 256 input binary features and three fully-connected layers. The three models differ by the number of neurons in the hidden layers: [32, 16, n]; [64; 32; n]; [128, 64, n] where $n = 10$ for the IoT use case and $n = 2$ for the Security use case.

We also include two additional columns (TCAM and SRAM) reporting the memory consumption for the TCAM-based and SRAM-based implementations. In the case of BNNs, there is only an SRAM-based implementation, as re-

ported in the paper in Section 3.2.

The results for the IoT use cases are reported in Table 5, while Table 6 reports the results for the Security use case.

A.2.2 Security Anomaly Detection without host features

For the Security use case, as mentioned earlier, we also run the classifier tests to check that the implications of removing the three additional non-flow level features is minimal for the BNN accuracy: the three BNN models described in the paper (32,16,2; 64,32,2; 128,64,2) achieve accuracy [0.9114, 0.9162, 0.9198] when including the 3 extra features, and [0.9106, 0.9164, 0.9201] when not including them (a difference of at most 0.1% point). The results for Decision Trees and Random Forests are instead more impacted, as shown in Table 7.

A.2.3 Confusion Matrices

Figures 19 and 20 report the confusion matrices for the IoT Traffic Classification and Security Anomaly Detection (with all features) use cases, respectively. The matrices have been normalized by dividing the counts by the sum of each row. For each use case we selected a single fold for each of the 9 representative models. In the 3x3 grid, each row contains a different type of model, i.e. Decision Trees (DT), Random Forests (RF) and Binary Neural Networks (BNN). For a given row, different columns contain an increasingly more complex model of a same type, e.g. a more deep tree-based model or a MLP with a larger number of neurons in the hidden layers.

The confusion matrices in the IoT use case confirm that small DTs and RFs fail to properly classify samples belonging to some classes. This is also a byproduct of using binary-decision trees, which fail to identify all of the 10 classes when so shallow. Performance improves as the complexity of the model increases. BNNs are instead able to classify almost all the classes even in the smallest configuration.

A.3 In-NIC Feature Extraction

As mentioned in the Section 6 of the paper, we implement two different features extraction strategies in both the Netronome NIC and NetFPGA. We give more details about these implementations in this subsection.

In both cases, we leverage the modern NIC's ability to host a large number of flow entries (several 10ks) in memory. For instance, both the Netronome and the NetFPGA are also equipped with relatively large DRAMs that can be leveraged to host very large flow tables.

A.3.1 Feature Extraction without connection tracking

The simpler feature extraction strategy keeps a hashtable with the active flows, and performs the following operations, on packet reception: (i) packet parsing to extract the needed
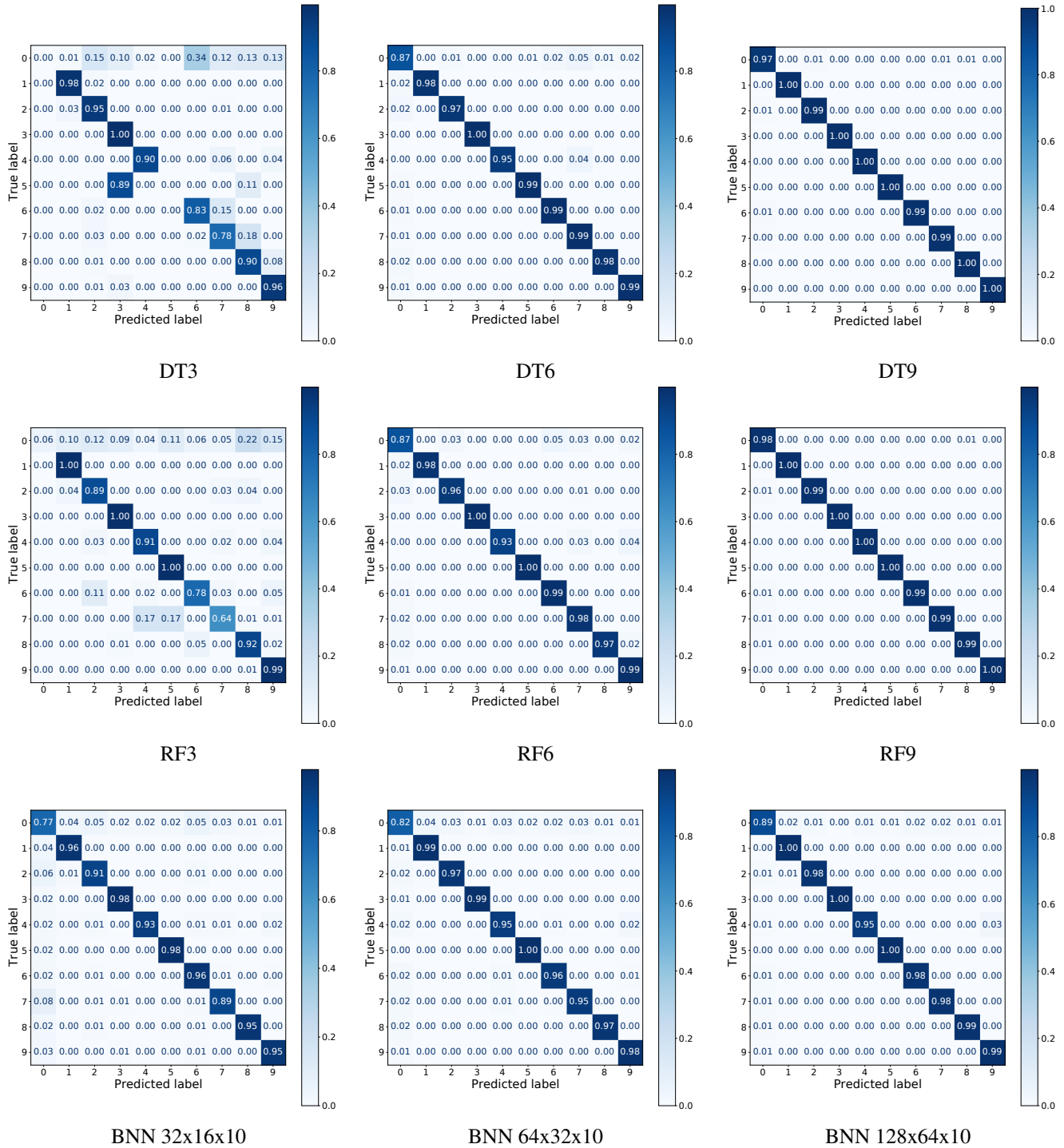
Figure 19: Confusion Matrices for the IoT use case

information, including the 5-tuple used as lookup key; (ii) lookup in the hashtable to retrieve the corresponding flow counters; (iii) update of the values to account for the new packet reception.

To keep the implementation as simple as possible, we do not perform any TCP connection tracking for TCP flows. To measure the flow features that we need for traffic analysis, in fact, in this simpler implementation it is enough to track the initial TCP handshake (e.g., to extract SYN-ACK RTTs). To measure flow duration, instead, we store the timestamp of the first packet of a flow (recognized by the absence of a flow entry in the flow hashtable) and check the timestamp of the
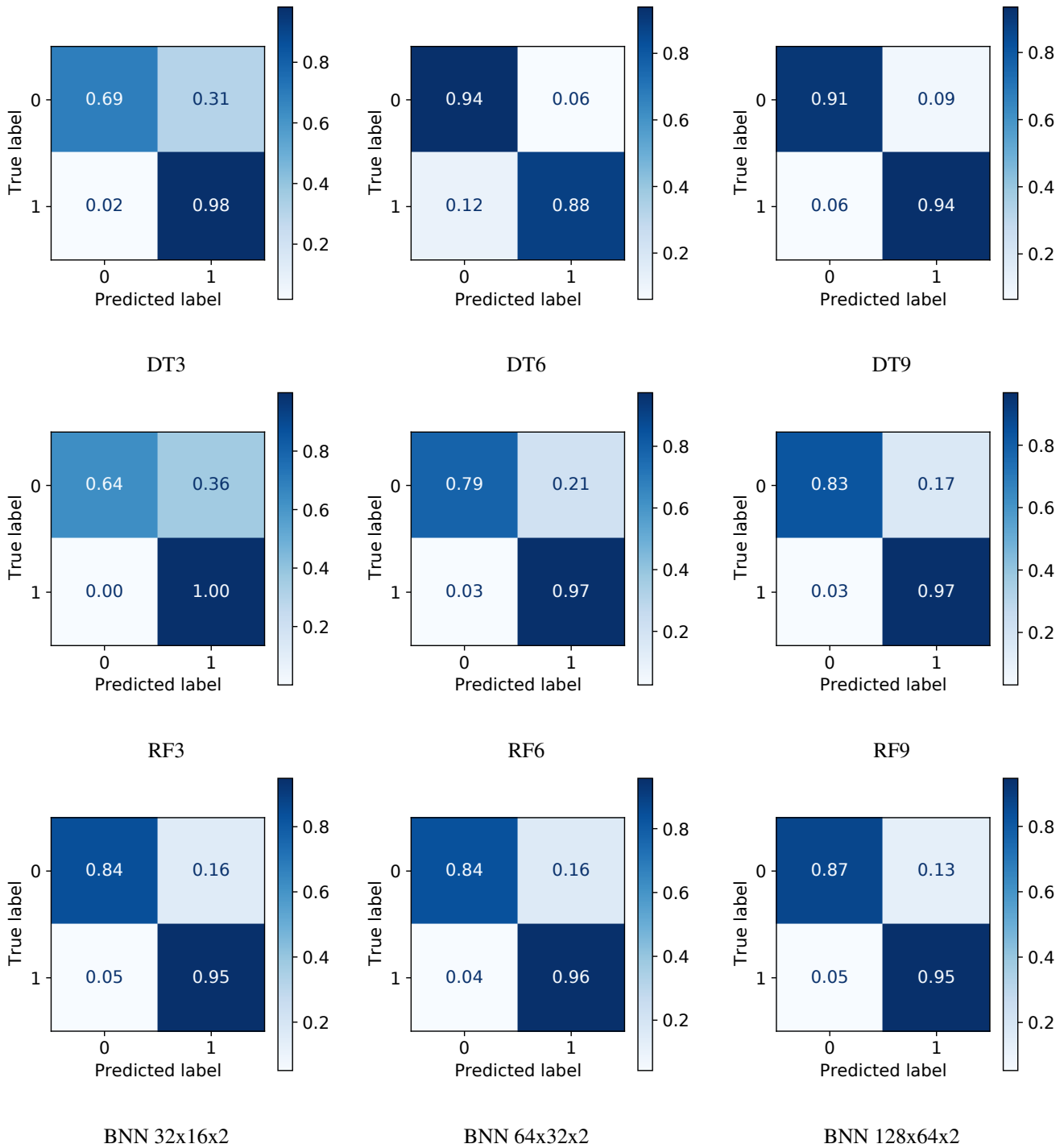
Figure 20: Confusion Matrices for the Security Anomaly Detection use case

last received flow's packet.

Flow entries are removed from the hashtable if no packets match them for a given amount of time. This is implemented as a lightweight task that can be performed lazily. For instance, when a new packet is received, if there is already an entry for the corresponding flow, but `current time -` `last packet timestamp > timeout value`, then the existing entry is discarded and the flow is considered as a new flow, and the received packet as the first packet of this flow. The timeout value should be configured depending on the deployment environment, taking into account the properties of the monitored traffic. For instance, in telecom operators
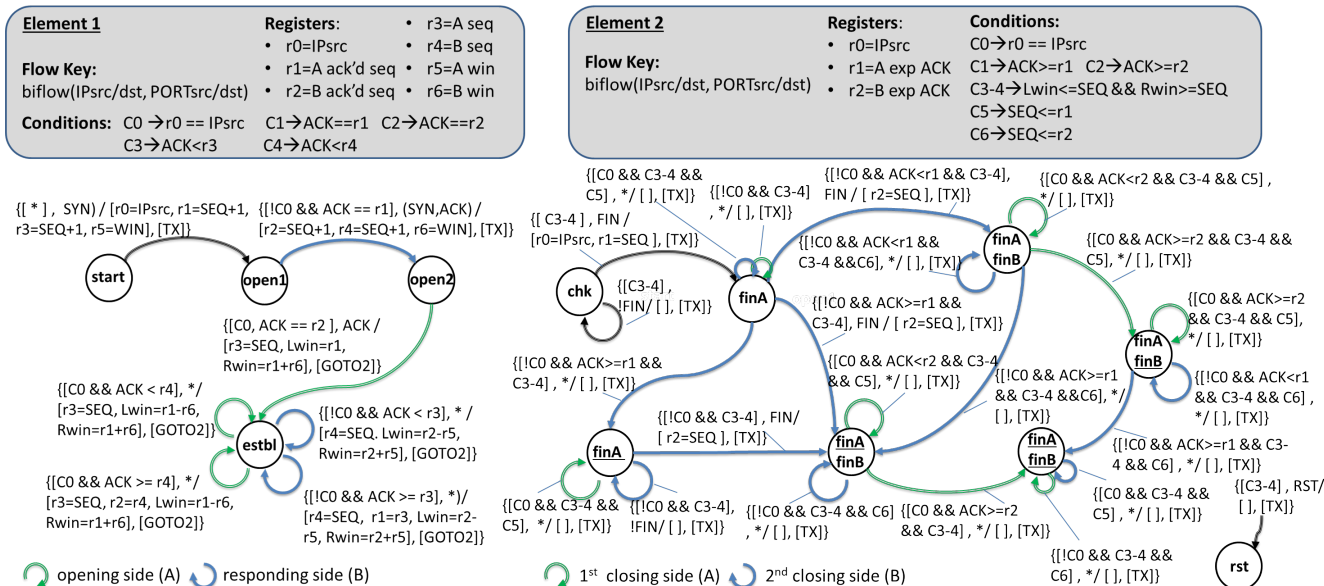
Figure 21: TCP Connection Tracking state machine, reported from [36]

A.3.2 Feature Extraction with connection tracking

The simpler implementation presented earlier is not safe in presence of misbehaving packets. For instance, an attacker may forge packets to impact the measured flow's features. This is possible, since the flow counters are only retrieved using the packet's 5-tuple, which in a general case may be e.g., forged. To avoid this class of issues, for TCP flows it is possible to perform TCP Connection Tracking. Connection tracking verifies that the flow's behavior is consistent with the TCP's state machine, and it includes fine granular per-packet checks, e.g., reading sequence and ack numbers.

We implement TCP connection tracking using the implementation presented in FlowBlaze [36], and the state machine is reported in Figure 21. Here, it should be noted that the state machine is in fact a sequential composition of two state machines. This is a by-product of using the FlowBlaze abstraction, which implements state machines in a sequence of stages that resemble a match-action pipeline similar to that of devices supporting the P4 language.

The two state machines are always executed in sequence, for each packet of an established connection. However, it is possible to identify different responsibilities of each of the two state machines. The first one tracks connection establishment, and computes the allowed sequence numbers (e.g., computing Rwin and Lwin). These values are *forwarded* to the second state machine that performs the actual checks, and which also implements the transitions to check the connection termination.

We implemented this connection tracking solution both in the NetFPGA and in the Netronome NIC. For the NetFPGA, we add two FlowBlaze stages in front of the N3IC design. These two stages are used to then to implement the state machine of Figure 21. For the Netronome, we implement the state machine of Figure 21 using micro-C, and extending the N3IC's Netronome firmware.

	Performance							Memory	
	Accuracy	Precision	Recall	FNR	FPR	F1-score	ROC-AUC	TCAM	SRAM
DT(3)	73.1 ± 0.1	61.0 ± 0.0	73.1 ± 0.1	26.9 ± 0.1	3.0 ± 0.0	65.8 ± 0.1	85.1 ± 0.0	119 B	40.2 kB
DT(6)	97.0 ± 0.1	97.0 ± 0.0	97.0 ± 0.1	3.0 ± 0.1	0.3 ± 0.0	97.0 ± 0.1	98.3 ± 0.0	1.3 kB	161.9 kB
DT(9)	99.4 ± 0.0	99.4 ± 0.0	99.4 ± 0.0	0.6 ± 0.0	0.1 ± 0.0	99.4 ± 0.0	99.7 ± 0.0	7.2 kB	170.2 kB
RF(3,5)	81.5 ± 0.3	83.4 ± 0.2	81.5 ± 0.3	18.5 ± 0.3	2.1 ± 0.0	77.5 ± 0.6	89.7 ± 0.2	595 B	200.8 kB
RF(6,5)	96.9 ± 0.2	97.0 ± 0.1	96.9 ± 0.2	3.1 ± 0.2	0.3 ± 0.0	96.9 ± 0.2	98.3 ± 0.1	6.4 kB	809.3 kB
RF(9,5)	99.4 ± 0.1	99.4 ± 0.1	99.4 ± 0.1	0.6 ± 0.1	0.1 ± 0.0	99.4 ± 0.1	99.7 ± 0.0	35.9 kB	851.0 kB
BNN [32,16,10]	92.4 ± 0.2	92.4 ± 0.3	92.4 ± 0.2	7.6 ± 0.2	0.8 ± 0.0	92.4 ± 0.2	95.8 ± 0.1	-	1.2 kB
BNN [64,32,10]	96.0 ± 0.1	96.0 ± 0.1	96.0 ± 0.1	4.0 ± 0.1	0.4 ± 0.0	96.0 ± 0.1	97.8 ± 0.1	-	2.5 kB
BNN [128,64,10]	97.4 ± 0.2	97.5 ± 0.2	97.4 ± 0.2	2.6 ± 0.2	0.3 ± 0.0	97.4 ± 0.2	98.6 ± 0.1	-	5.5 kB

Table 5: IoT dataset

	Performance							Memory	
	Accuracy	Precision	Recall	FNR	FPR	F1-score	ROC-AUC	TCAM	SRAM
DT(3)	88.0 ± 0.2	85.3 ± 0.2	98.2 ± 0.0	1.8 ± 0.0	30.1 ± 0.4	86.0 ± 0.2	84.1 ± 0.2	102 B	173.3 kB
DT(6)	90.3 ± 0.1	96.3 ± 0.1	88.2 ± 0.2	11.8 ± 0.2	5.9 ± 0.1	89.8 ± 0.1	91.1 ± 0.1	677 B	18.9 MB
DT(9)	92.5 ± 0.2	95.0 ± 0.1	93.2 ± 0.2	6.8 ± 0.2	8.7 ± 0.3	91.9 ± 0.2	92.2 ± 0.2	3.4 kB	19.9 MB
RF(3,5)	87.3 ± 0.2	83.4 ± 0.3	99.9 ± 0.0	0.1 ± 0.0	35.2 ± 0.6	84.8 ± 0.3	82.4 ± 0.3	512 B	866.4 kB
RF(6,5)	90.5 ± 0.5	88.6 ± 1.3	97.7 ± 0.9	2.3 ± 0.9	22.2 ± 2.9	89.2 ± 0.7	87.7 ± 1.0	3.4 kB	94.7 MB
RF(9,5)	92.3 ± 0.3	92.7 ± 1.3	95.5 ± 1.1	4.5 ± 1.1	13.3 ± 2.7	91.6 ± 0.4	91.1 ± 0.8	16.9 kB	99.3 MB
BNN [32,16,2]	91.1 ± 0.1	91.4 ± 0.6	95.1 ± 0.6	4.9 ± 0.6	15.9 ± 1.2	90.2 ± 0.2	89.6 ± 0.4	-	1.2 kB
BNN [64,32,2]	91.6 ± 0.1	92.4 ± 0.6	94.7 ± 0.6	5.3 ± 0.6	13.8 ± 1.2	90.8 ± 0.1	90.4 ± 0.3	-	2.5 kB
BNN [128,64,2]	92.0 ± 0.1	92.8 ± 0.4	94.8 ± 0.4	5.2 ± 0.4	13.0 ± 0.8	91.2 ± 0.1	90.9 ± 0.2	-	5.4 kB

Table 6: Security dataset

	Performance							Memory	
	Accuracy	Precision	Recall	FNR	FPR	F1-score	ROC-AUC	TCAM	SRAM
DT(3)	88.0 ± 0.2	85.3 ± 0.2	98.2 ± 0.0	1.8 ± 0.0	30.1 ± 0.4	86.0 ± 0.2	84.1 ± 0.2	102 B	173.3 kB
DT(6)	89.9 ± 0.5	87.6 ± 1.4	98.2 ± 1.2	1.8 ± 1.2	24.7 ± 3.5	88.5 ± 0.7	86.8 ± 1.1	677 B	18.9 MB
DT(9)	91.2 ± 0.1	90.6 ± 0.5	96.2 ± 0.8	3.8 ± 0.8	17.7 ± 1.1	90.2 ± 0.1	89.2 ± 0.2	3.4 kB	19.9 MB
RF(3,5)	87.3 ± 0.2	83.4 ± 0.2	100.0 ± 0.0	0.0 ± 0.0	35.2 ± 0.6	84.8 ± 0.3	82.4 ± 0.3	512 B	866.4 kB
RF(6,5)	89.6 ± 0.4	86.7 ± 0.8	98.9 ± 0.8	1.1 ± 0.8	26.9 ± 2.2	88.0 ± 0.5	86.0 ± 0.7	3.4 kB	94.7 MB
RF(9,5)	91.4 ± 0.3	90.3 ± 0.7	97.0 ± 0.6	3.0 ± 0.6	18.5 ± 1.5	90.4 ± 0.4	89.3 ± 0.5	16.9 kB	99.3 MB
BNN [32,16,2]	91.1 ± 0.2	91.3 ± 0.6	95.1 ± 0.7	4.9 ± 0.7	16.1 ± 1.3	90.1 ± 0.2	89.5 ± 0.3	-	1.2 kB
BNN [64,32,2]	91.6 ± 0.1	92.7 ± 0.2	94.4 ± 0.3	5.6 ± 0.3	13.3 ± 0.5	90.9 ± 0.1	90.6 ± 0.1	-	2.5 kB
BNN [128,64,2]	92.0 ± 0.2	93.0 ± 0.5	94.6 ± 0.4	5.4 ± 0.4	12.6 ± 0.9	91.3 ± 0.2	91.0 ± 0.3	-	5.4 kB

Table 7: Security dataset when not including the three host features

Elixir: A High-performance and Low-cost Approach to Managing Hardware/Software Hybrid Flow Tables Considering Flow Burstiness

Yanshu Wang[★], Dan Li[★], Yuanwei Lu[†], Jianping Wu[★], Hua Shao[★], Yutian Wang[★]

[★]*Tsinghua University,* [†]*Tencent*

Abstract

Hardware/software hybrid flow table is common in modern commodity network devices, such as NFV servers, smart NICs and SDN/OVS switches. The overall forwarding performance of the network device and the required CPU resources are considerably affected by the method of how to split the flow table between hardware and software. Previous works usually leverage the traffic skewness for flow table splitting, *e.g.* offloading top 10% largest flows to the hardware can save up to ∼90% CPU resources. However, the widely-existing *bursty flows* bring more challenges to flow table splitting. In particular, we need to identify the proper flows and proper timing to exchange the flows between hardware and software by considering flow burstiness, so as to maximize the overall performance with low overhead.

In this paper we present Elixir, a high-performance and low-cost approach to managing hardware/software hybrid flow tables on commodity devices. The core idea of Elixir includes three parts, namely, combining sampling-based and counter-based mechanisms for flow rate measurement, separating the replacement of large flows and bursty flows, as well as decoupling the flow rate identification window and the flow replacement window. We have implemented Elixir prototypes on both Mellanox ConnectX-5 NIC and Barefoot Wedge100BF-32X/65X P4 Switch, with a software library on top of DPDK. Our experiments based on real-world data traces demonstrate that, compared with the state-of-the-art solutions, Elixir can save up to ∼50% software CPU resources while keeping the tail forwarding latency ∼97.6% lower.

1   Introduction

With more and more packet header fields taken as the input for forwarding rules, the size of flow table in modern network devices grows rapidly [3, 4, 59, 64, 71]. Although hardware has fast forwarding speed, the hardware on-chip memory (typically containing <6M flows [13, 49, 50]) usually cannot serve all the concurrent flows (typically with an order of O(10M) [31]). On the contrary, the software has large memory

and limited forwarding capacity. As a result, many commodity network devices, such as NFV servers, smart NICs and SDN/OVS switches, take a hardware/software hybrid way to manage the large flow table [4, 15, 16, 61]. By using this kind of hybrid flow table, the typical packet forwarding process is as follows. Upon receiving traffic, the hardware extracts certain fields from the received packet's header. If an entry in the hardware flow table is hit, the action associated with the entry is executed on the packet; otherwise, the packet is forwarded to CPU to match the software flow table.

In this scenario, it is important to figure out how to split the flow table between hardware and software. The splitting method not only affects the forwarding performance, but also determines the CPU resources reserved for software forwarding, which is of particular importance for cloud environment. Previous works usually leverage the traffic skewness for flow splitting [5, 15, 18, 53, 55, 61, 73], *e.g.* offloading top 10% largest flows to the hardware can save up to ∼90% CPU resources. However, the wide existence of *bursty flows* [19, 25, 33, 48, 52] brings more challenges to be addressed to flow table splitting, for maximizing the overall forwarding performance with low overhead.

First, how to accurately measure all the flow rates with low overhead on commodity devices?

The dynamic flow replacement between hardware and software requires a timely and accurate identification of all the flow rates including bursty ones. As the hardware flows bypass the software, the measurement cannot be done solely by software. Commodity hardware devices, such as Mellanox NICs and P4 Switches, support both putting a hardware counter to every flow and sampling the hardware traffic to software. The cost of using hardware counters for flow rate measurement is very high (more discussion in Section 2). If sampling a portion of the hardware traffic to software for measurement, a too large sampling rate wastes many CPU resources while a too small sampling rate might miss bursty flows, which can result in packet loss. Given a certain sampling rate, we should also set a proper window size for accurate flow rate identification. Therefore, we need a method to

accurately measure all the flow rates with low overhead on commodity devices, so as to identify the appropriate set of flows for replacement.

Second, how to set the proper timing for flow replacement between hardware and software?

Due to traffic dynamics, some flow table entries in hardware and software should be exchanged over time. Conventional approaches make periodic replacement, *i.e.* if a software flow's rate becomes larger than a hardware flow's rate within a time window, the two flows are exchanged with each other. However, in practice we find it difficult to set the proper window size for periodic replacement. In order to timely offload software flows whose rates turn large, the window should be as small as possible. However, the small window will lead to frequent flow table replacement. In the high-speed packet forwarding scenario, too frequent flow table replacement will cause considerable performance degradation, in both hardware and software. On the other side, if we set a large replacement window to mitigate forwarding performance degradation, bursty flows with lower average rate than stable flows might be kept in software. To prevent packet loss of these bursty flows, we have to provision much more CPU resources for the peak rate, which is usually several orders of magnitude higher than the average rate. It is a considerable waste of CPU resources. Therefore, we have to solve this dilemma of setting the proper flow replacement window between hardware and software.

In this paper, we propose Elixir, a high-performance and low-cost approach to managing hardware/software hybrid flow tables on commodity devices, by taking flow burstiness into account. Elixir deals with the aforementioned challenges as follows.

First, Elixir combines both sampling-based and counter-based mechanisms for the rate measurement of large flows and bursty flows, respectively. For large flows, packets are sampled from hardware to software with a low sampling rate, so as to reduce the processing overhead on CPU. For bursty flows that are easy to be missed by low-rate sampling, hardware counters are put to each flow, since the number of concurrent bursty flows is not quite large (based on our observation from real traces). By this method, Elixir leverages the benefit of both hardware and software solutions to get the balance between measurement accuracy and measurement overhead.

Second, Elixir separates the replacement processes of large flows and bursty flows. For large flows, due to their stable rate changing pattern over time, Elixir periodically exchanges large flows between hardware and software. A relatively large replacement window is used to offload large flows with the highest average rates during this window, so as to avoid the throughput degradation caused by too frequent replacement. Meanwhile, for bursty flows, as they appear in the system at irregular basis, Elixir leverages an event-driven process to offload them to hardware, *i.e.* bursty flows are offloaded

immediately when they are detected. By observing that the size of software queue increases dramatically when a bursty flow comes, Elixir uses it as a signal to trigger the bursty flow offloading process. In this way, Elixir makes the tradeoff between burst-aware offloading and forwarding performance degradation caused by frequent replacement.

Third, Elixir decouples the flow rate identification window and the flow replacement window. Previous works [6, 21, 66] make no distinction between the two windows. In principle, the flow rate identification window is determined by traffic characteristics while the flow replacement window is determined by hardware/software system limitations. Consequently, using one size for the two windows may either sacrifice the flow rate identification accuracy, or lead to lagged flow replacement. Elixir explicitly decouples the two and independently decides the proper sizes for them. For large flows, the flow replacement window is set to the minimum replacement decision interval which brings affordable impact on the forwarding performance; while the flow rate identification window is set to the minimum window which can accurately identify flow rates, since the sampled traffic may cause inaccurate rankings of large flow rates. For bursty flows, the flow rate identification window is set to be small enough to catch the flow burstiness and they are offloaded immediately once detected. By the decoupling, Elixir achieves the tradeoff between timely flow replacement and accurate flow identification.

We have implemented Elixir prototypes on both Mellanox ConnectX-5 NIC and Barefoot Wedge100BF-32X/65X P4 Switch, with a software library on top of DPDK. We run experiments based on the real-world traces from cloud gateways of an Internet content provider. The results show that Elixir can save up to ∼50% software CPU resources while keeping the tail forwarding latency ∼97.63% lower compared with TFO [61] and ∼97.61% lower compared with LFP [15].

2 Motivation and Challenges

In this section, we first describe the findings from traffic measurement of typical cloud gateways, then we present the design challenges for a hardware/software hybrid flow table management solution.

2.1 Traffic Measurement of Cloud Gateways

We examine three cloud gateways of an Internet content provider. The gateways run on commodity servers using hardware/software hybrid flow tables. We collect traffic traces from all the three gateways. The data path of these gateways manipulates packet headers and forwards them with tunnels, *e.g.* GRE or VxLAN [17, 43]. For each gateway, we have collected the real-time packet-level traces of a work day. In what follows we describe our key findings from the traffic traces of the three gateways. Due to page limits, we only present the results for one gateway, since the traffic characteristics of the other two are quite similar.
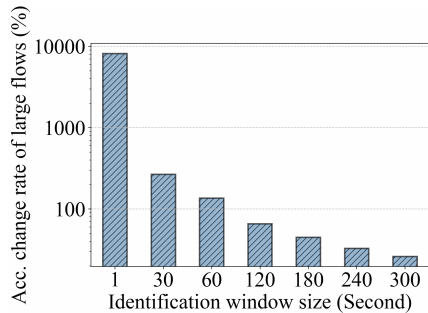
Figure 1: Accumulated change rate of large flows against different flow rate identification window sizes.

(a) Distribution of the burst ratio (peak/average rate).

(b) Distribution of the numbers of concurrent bursty flows.

Figure 2: The characteristics of bursty flows in the cloud gateway trace.

(a) Queue size and latency.

(b) Packet loss rate against different software-forwarding CPU cores.

Figure 3: The cost of forwarding the bursty flow and the stable flow by software.

Large flows constantly change over time: The rates of flows dynamically change over time. For the traces, we set the flow rate identification window to different sizes, *i.e.* from 1 second to 300 seconds. For each window size, we measure the top 10% flows with the largest flow rates in every time window (referred to as *large flows*), and count the number of changed large flows between neighboring windows within a 10-minute period. Then we divide this number by the total number of the flows to calculate the accumulated change rates of large flows. Fig. 1 shows the accumulated change rates of large flows within the 10-minute period. We find that large flows constantly change over time. It indicates that, in order to efficiently manage the hardware/software hybrid flow table and offload large flows to hardware, the flow table entries in hardware and software need to be periodically exchanged. Moreover, given a fixed time period, a smaller flow rate identification window leads to higher accumulated changes of large flows between neighboring windows.

Bursty flows are common: Previous works on flow table splitting primarily consider offloading *large flows*, with a focus on the average flow rates during a time period. They pay little attention to *bursty flows*, of which the flow rates surge to a high value quickly and last for a short time period before dropping to a low rate. Bursty flows are very common in both backbone and datacenter networks [19, 25, 33, 48, 48, 52], which may come from bursty applications (such as video application), TCP incast, or batching operation of network stacks [48].

Conceptually, a bursty flow can be or not be a large flow; besides, a large flow can be either a bursty large flow during some periods while a stable large flow during other periods. In order to study the impact of bursty flows, we measure the rate changing pattern for each individual flow in the cloud gateway trace.

The results show that bursty flows are also quite common in the trace. We use the ratio of a flow's peak rate (in a second) over its average rate, named *burst ratio*, to describe the level of flow burstiness. We depict the CDF of burst ratio in Fig. 2(a). As shown in the figure, ~80% of the flows have a >20 burst ratio, with the maximum ratio as high as 80. By examining the data, we find that the peak rate duration of most bursty flows is very short, *e.g.* several seconds. Moreover, we make
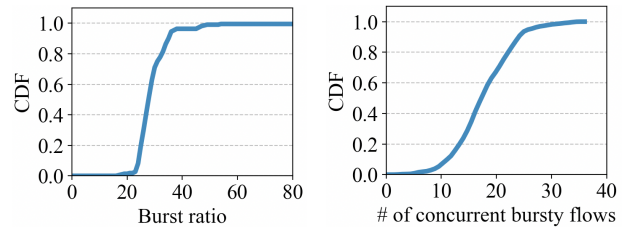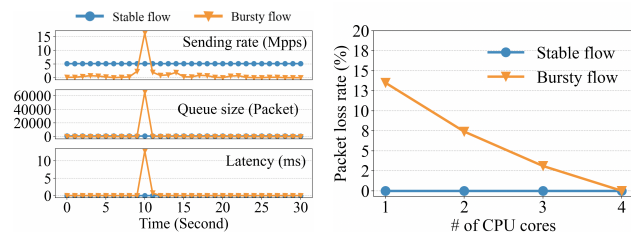
statistics about the distribution of the numbers of concurrent bursty flows (using bursty flows with a >10 burst ratio in the case), shown in Fig. 2(b). We find that, although bursty flows are common, their bursty times are usually not overlapped. In other words, the number of concurrent bursty flows is limited, *e.g.* with the largest number as 36 in our trace.

Bursty flows require more software forwarding resources: We further carry on experiments to quantitatively compare the cost of forwarding bursty flows and stable flows by software. We use three servers, one as the sender, one as the receiver and the third as the software forwarder. We purposely generate a stable flow and a bursty flow at the sender. Notably, the average rate of the stable flow is 5 times higher than that of the bursty flow. Each flow is only forwarded by the software.

We first use one CPU core at the software forwarder. We separately run the two flows, and record the end-to-end latency as well as the queue size at the software forwarder. The results are shown in Fig. 3(a). It indicates that, for the stable flow and the stable periods of the bursty flow, the queue size is small and the end-to-end latency is low; but at the peak rate period of the bursty flow, the queue size and the end-to-end latency sharply increase, by a maximum of ~30 and ~28 times respectively.

Then we use different numbers of CPU cores at the software forwarder to forward the two flows and measure the packet loss rates. As demonstrated by Fig. 3(b), although the average rate of the bursty flow is only one fifth of the stable flow, the software forwarding resources required by the bursty flow are 4 times of the stable flow to avoid packet loss. Specif-

ically, to prevent packet loss from occurrence, at least 4 CPU cores are required at the software forwarder for the bursty flow, while only 1 CPU core is needed for the stable flow. In conclusion, since the CPU resources should be reserved for the peak rate of a flow instead of the average rate to prevent packet loss, in practice bursty flows occupy remarkably more software forwarding resources than stable flows.

2.2 Design Challenges

Previous works [15, 55, 61, 73] on managing hardware/software hybrid flow tables usually leverage the traffic skewness for flow table splitting, *i.e.* offloading a small portion of the largest flows to the hardware can save most of the CPU resources. However, by considering flow burstiness, more challenges have to be addressed, so as to maximize the overall performance with low overhead.

Challenge 1: how to accurately measure all the flow rates with low overhead on commodity devices?

An accurate and low-cost approach to measuring all the flow rates is critical in hybrid flow table management on commodity devices. Note that hardware can see all the flows but software can only see the software-forwarded flows, hence the flow rate measurement cannot be done without the support of hardware. Knowledgeable readers may consider building a sketch data structure [12, 23, 39, 41, 68–70, 72, 74] in hardware to measure all the flow rates. However, commodity hardware does not support this kind of functionality yet. Besides, a sketch for so many flows consumes too many resources in hardware, which can be saved for storing more forwarding rules.

Modern commodity hardware devices, such as Mellanox NICs and P4 Switches, support both putting a hardware counter to every flow and sampling the hardware traffic to software (Mellanox plans to support the sampling functionality in the new release). Hence, one candidate solution is to place a hardware counter for each flow and use the counters to accurately measure each flow's rate. However, there are two problems for this method. First, setting a counter for each flow occupies additional hardware resources. Based on our measurement, hardware counters result in ∼ 20% less space for hardware (NIC and switch) forwarding rules, which will result in much more traffic forwarded to software and much more CPU resources consumed consequently. Second, the speed for software to read the counters from commodity hardware devices is slow, *e.g.* the speed is about 20k rules per second for Mellanox ConnectX-5 NIC. It means that several seconds are required for the software to read all the hardware counters, which is too slow for timely flow replacement.

The final optional solution is to sample a portion of hardware traffic to software and use the software to measure all the flow rates. However, as the peak rate duration of bursty flows is quite short, a high sampling rate is required in order not to miss them; otherwise, bursty flows might not be correctly identified to offload from software to hardware, or might be
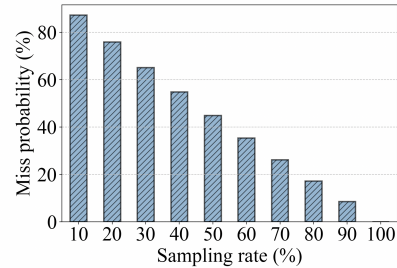

Figure 4: Miss probability of bursty flows against different sampling rates.
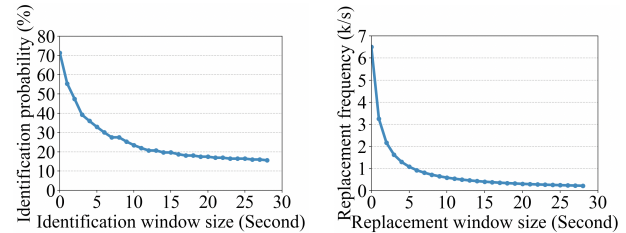
(a) Identification probability of bursty flows.

(b) Replacement frequency of large flows.

Figure 5: Identification probability of bursty flows and replacement frequency of large flows against different time window sizes.

replaced from hardware to software by mistake. As shown in Fig. 4, in the cloud gateway trace, a 20% sampling rate will lead to a probability of 76% to miss bursty flows. However, if we set a high sampling rate, much more CPU resources will be occupied. Hence, it is a dilemma how to set the proper sampling rate if we use a sampling-based measurement approach.

As a result, we have to design an accurate and low-cost flow measurement method, by overcoming the problems above.

Challenge 2: How to set the proper timing for flow replacement between hardware and software?

Conventional approaches make periodic flow replacement between hardware and software, *i.e.* if a software flow's rate becomes larger than that of a hardware flow within a periodic time window, the two flows are exchanged with each other. Following this approach, as discussed above, if we set a large time window to replace flows between hardware and software, bursty flows might have lower average rate than stable flows during this window and thus be kept in software. It will result in more CPU resources reserved for software forwarding. Based on the cloud gateway trace, we make further analysis to measure the probability for bursty flows to rank higher than stable flows, *i.e.* bursty flows being identified as large flows to offload by previous solutions, against different identification window sizes. As shown in Fig. 5(a), for bursty flows to rank higher than stable flows with >60% probability, the identification window should be as small as <2 seconds.

On the other side, if we set a small time window for flow replacement, it usually means much more flows to exchange between hardware and software during a fixed time period
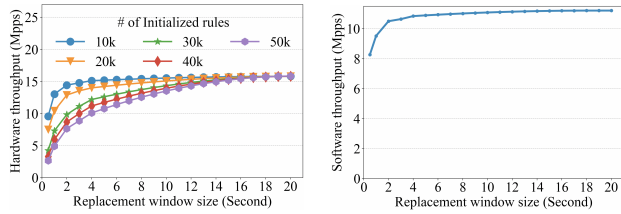
(a) Hardware throughput against different replacement window sizes and numbers of initialized rules.

(b) Software throughput against different replacement window sizes.

Figure 6: Forwarding speed will be degraded by frequent flow replacement between hardware and software.

(refer to Fig. 1). In other words, a small flow replacement window results in high replacement frequency. It is validated by Fig. 5(b), which shows the replacement frequency against different replacement window sizes based on the cloud gateway trace, *e.g.* a 2-second replacement window results in a replacement frequency of 3.25k/s.

Unfortunately, high replacement frequency causes forwarding speed degradation, in both hardware and software. For hardware, it cannot keep forwarding traffic at a high rate with too frequent rule replacement, due to the lock mechanism used. We measure the forwarding speed under different replacement frequencies using a commodity Mellanox ConnectX-5 NIC. As shown in Fig. 6(a), with 50k initial hardware flows updated at a replacement frequency of 3.25k/s (2-second time window), the NIC throughput drops by 50%. For Barefoot P4 Switches, today the vendors limit the maximum replacement frequency to 2k/s, which can merely avoid obvious performance degradation during replacement. For software, frequent replacement leads to cache contention and results in many cache misses, which can also degrade the forwarding throughput. Specifically, in our trace, the software throughput drops by 16% at a replacement frequency of 3.25k/s, as shown in Fig. 6(b).

Therefore, if a conventional periodic replacement method is used, we have to resolve the dilemma of how to set the proper size of the time window for high-performance hybrid flow table management.

3   Design

In this section, we elaborate the design of Elixir. We first describe the design overview, and then separately present the design details of large flow replacement and bursty flow replacement.

3.1   Design Overview

Elixir takes the following key ideas to address the challenges discussed in Section 2.

First, Elixir uses different methods for measuring the rates of bursty flows and large flows. As large flows usually comprise of many packets, a small portion of sampled traffic can give relatively accurate information about flow characteristics.

Motivated by that, Elixir samples hardware traffic to software with a low sampling rate, so as for large flow identification. Observing that the number of concurrent bursty flows is small (refer to Fig. 2(b)), Elixir associates each bursty flow with a hardware counter and the software polls the counters for flow rate measurement.

As different measurement techniques are used for large flows and bursty flows, Elixir separates the hardware flow table into two disjoint areas, *i.e.* a large flow area and a bursty flow area. Large flows and bursty flows are inserted into corresponding areas accordingly. It is worth noting that, if a flow is both a large flow and a bursty flow (as discussed in Section 2), we insert the flow's forwarding rule to the corresponding hardware area where the replacement mechanism is firstly triggered for the flow. In principle, no matter in which hardware area is the forwarding rule stored, the flow will be forwarded by hardware and will not cause packet loss in software. Hence, when the flow turns from a bursty large flow to a stable large flow or vice versa, it is unnecessary to move the flow to other hardware area.

By this method, Elixir leverages the benefits of both hardware and software solutions to get the balance between measurement accuracy and measurement overhead.

Second, Elixir separates the replacement of large flows and bursty flows, due to their distinct characteristics. As shown in Fig. 1, large flows constantly change over time. If not adjusted periodically, throughput of software traffic may gradually increase, leading to higher CPU usage. It indicates that we need a periodic process to exchange large flows between hardware and software, and the replacement window can be set relatively larger, so as to bring affordable impact on the overall forwarding performance.

In contrast, bursty flows appear in the system at irregular basis, and cause a sharp increase of the queue size and forwarding latency. If not offloading software-forwarded bursty flows quickly, it may result in severe packet loss. As a result, we need to detect the existence of bursty flows quickly and offload them as soon as possible, which is an event-driven procedure.

Overall, Elixir handles large flows and bursty flows separately with distinct methods, *i.e.* large flows are offloaded periodically while bursty flows are offloaded immediately once detected. In this way, Elixir makes the tradeoff between burst-aware offloading and forwarding performance degradation caused by frequent replacement.

Third, Elixir decouples the flow rate identification window and the flow replacement window. Flow rate identification window (referred to as *identification window* for short in the rest of this paper) is the time window to measure the flow throughput, *i.e.* flows that generate the most traffic within the window are identified as the large flows. As a result, the size of the identification window should be determined by traffic pattern. Differently, flow replacement window (referred to as *replacement window* for short in the rest of this paper) is
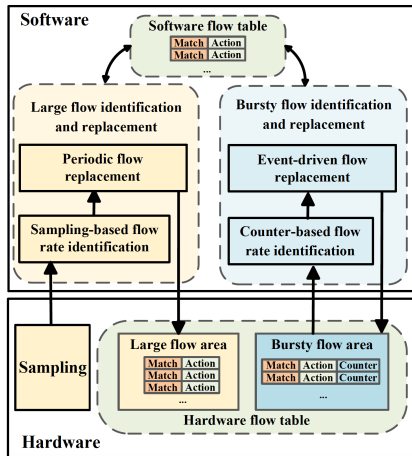
Figure 7: Architecture overview of Elixir.

Figure 8: Identification accuracy against different sampling rates and identification window sizes.

Figure 9: Percentage of software traffic increase against different identification window sizes and replacement window sizes.

the time window between two adjacent hardware/software flow replacement decisions, which has close relationship with hardware/software system characteristics, *i.e.* too frequent flow replacement causes the performance degradation of both hardware and software. Consequently, using one size for the two windows may either sacrifice the flow rate identification accuracy or lead to lagged flow replacement.

Based on the observation, Elixir explicitly decouples the identification window and the replacement window. Specifically, for large flows, the replacement window is set to the minimum replacement decision interval which brings affordable impact on the forwarding performance; while the identification window is set to the minimum window which can accurately identify flow rates, since the sampled traffic may cause inaccurate rankings of large flow rates. For bursty flows, the identification window is set to be small enough to catch the flow burstiness, and they are offloaded immediately once detected.

By this decoupling, Elixir achieves the tradeoff between timely flow replacement and accurate flow identification.

Based on the key ideas above, the architecture overview of Elixir is shown in Figure 7. Two different policies are separately run for large flows and bursty flows. For large flows, periodic replacement policy is used, with sampling-based rate identification and a relatively large replacement window; for bursty flows, event-driven replacement policy is adopted, with counter-based rate identification and the software queue size as the replacement signal.

3.2 Periodic Large Flow Replacement

As aforementioned, Elixir leverages a periodic procedure to exchange large flows between hardware and software, and the rates of large flows are identified using sampled traffic.

Setting the sampling rate and the identification window size: When sampling the hardware traffic to software, in order to reduce the computation and storage overhead in software, a low sampling rate is required. To further reduce
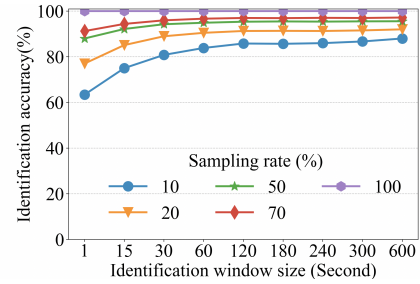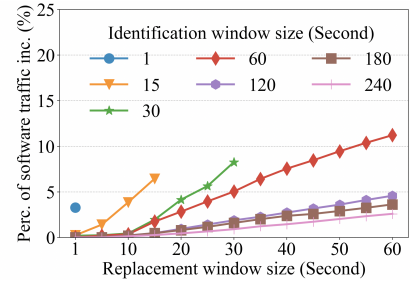
the hardware/software communication cost, the payload of every packet is cut and only the packet header is delivered from hardware to software. Since we use a low sampling rate, a flow's ranking in the sampled traffic may be different from its ranking in the actual traffic, which leads to inaccurate selection of flows to replace. Generally, a higher sampling rate or a larger identification window not only results in more accurate flow rate rankings as it sees more packets, but also means that more CPU and memory resources are required for flow rate identification in software. In a practical system, the sampling rate and the identification window size should be set by taking multiple factors into account, including CPU overhead, memory overhead and identification accuracy.

For the cloud gateway trace, Fig. 8 shows the rate identification accuracy [1] against different sampling rates and identification window sizes. Based on the results, we set the sampling rate to 20% and the identification window size to 30 seconds, which can achieve an identification accuracy of ~90%.

Setting the replacement window size: As aforementioned, Elixir decouples the identification window and the replacement window. When setting the replacement window size for large flows, if using a large window, some software flows which turn large cannot be timely offloaded, and the software-forwarded traffic may increase during the window, which causes more CPU resource consumption. If setting a small window, as shown in Fig. 6(a) and Fig. 6(b), the high replacement frequency may cause considerable forwarding performance degradation in both hardware and software.

---

[1] The rate identification accuracy is defined as the ratio of the number of top 10% largest flows that are correctly identified by our method over the total number of top 10% largest flows.

Figure 10: Data structures used in large flow identification and replacement algorithm.

Therefore, when setting the replacement window size, there should be a tradeoff between timely offloading and forwarding performance degradation caused by frequent replacement.

Ideally, if we can replace large flows as quickly as possible, *i.e.* immediately offloading a software flow when its rate becomes larger than a hardware flow, we can get *the smallest software traffic*. In practice, a large replacement window will cause lagged offloading and thus the actual software traffic will be larger than the smallest one. We meas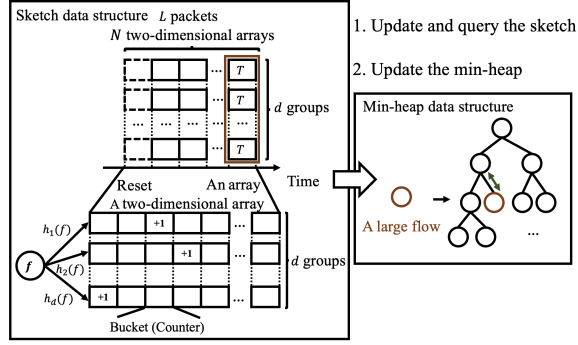ure the *percentage of traffic increase* in software against different identification window sizes and replacement window sizes, as shown in Fig. 9. It is worth noting that, under different identification window sizes, the same replacement window may also lead to different software traffic. For the identification window size of 30 seconds we choose for the cloud gateway trace, the additional software traffic is negligible when the replacement window is smaller than 10 seconds.

From Fig. 6, we find that when the replacement window is smaller than 10 seconds, the hardware and software forwarding speeds will degrade significantly. To make the tradeoff between the two factors, we can set the replacement window size to 10 seconds for the cloud gateway trace.

Identification and replacement algorithm for large flows: Based on the window setting rules discussed above, in a practical system the replacement window can be either smaller or larger than the identification window. In most cases when the replacement window is smaller than the identification window (*e.g.* for the cloud gateway trace), the identification window is actually a sliding window with the replacement window as the sliding step size. When the replacement window is larger than the identification window, which should be rare cases, we increase the identification window to the replacement window, since more information about the traffic results in more accurate flow rate rankings.

As shown in Fig. 10, we employ sketch and min-heap data structures to estimate the sizes of flows[2] received in an identification window. Assuming L is the identification window size and T is the replacement window size, there is $N = \lceil \frac{L}{T} \rceil$. Elixir uses N two-dimensional arrays to record a flow's re-

---

[2]The flow rate is obtained via dividing the flow size by the identification window size.

ceived packets in N sub-windows respectively, with the sub-window size set to the replacement window size T. Each two-dimensional array consists of d groups of buckets. A sampled packet is hashed into one of the buckets in different groups of the same two-dimensional array in the sketch structure, and each bucket contains a counter which will be increased by one upon packet arrival. The flow size can be calculated from the sums of all the corresponding counters in each sub-window. The minimum value of the counter sums associated with a flow is considered as the flow size. All the flows are then fed into a min-heap data structure for sorting, which yields the largest flows. Of course, if $N = 1$, which means that the replacement window is larger than or equal to the identification window, there is only one array in the identification window.

Algorithm 1 The identification and replacement algorithm for large flows.

Input: The identification window L, the replacement window T, the flow f of the incoming packet, a set of hash functions $h_j(j = 1, 2, 3...d)$, an array A[.][.][.], the current replacement window t initialized as 0, and a min-heap *min_heap*.

1: $N \leftarrow \lceil \frac{L}{T} \rceil$
2: **for** j=1 to d **do**
3:     A[j][$h_j(f)$][t]++
4: **end for**
5: flow_size $\leftarrow 0$
6: **for** j=1 to d **do**
7:     $C = \sum_{t=1,2,...,N}$ A[j][$h_j(f)$][t]
8:     **if** C < flow_size **then**
9:         flow_size = C
10:     **end if**
11: **end for**
12: **if** $f \in$ min_heap **then**
13:     min_heap[f] $\leftarrow$ flow_size
14: **else if** min_heap has empty buckets **then**
15:     min_heap.insert(f, flow_size)
16: **else if** flow_size > min_heap$_{min}$ **then**
17:     min_heap.delete_root()
18:     min_heap.insert(f, flow_size)
19: **end if**
20: **if** (time.now() $-$ pre_time_r) $\geq T$ **then**
21:     pre_time $\leftarrow$ time.now()
22:     current_flows $\leftarrow$ min_heap.all_flows
23:     replace_rules(previous_flows, current_flows)
24:     previous_flows $\leftarrow$ current_flows
25:     t $\leftarrow$ (t + 1) mod N
26:     min_heap.reset()
27:     A[.][.][t] $\leftarrow 0$
28: **end if**

The flow replacement operation is enforced at every T time. When flow replacement is finished, the identification window moves over T time and forgets the counters for the oldest replacement window (if the identification window is smaller than or equal to the replacement window, all the counters are reset). As there are several groups of counters, a naive

design would require a touch at each counter in each group when resetting the counters, which will heavily decrease the processing performance. Instead, Elixir arranges the counters of the same replacement window into an array. Thus each time the identification window moves, only the array which records the oldest counters needs to be reset, which is a piece of continuous memory and can be processed fast. When the identification window updates, the heap which keeps the rankings of the large flows is rebuilt using the latest packet counters. We use a bloom filter [7] to record the offloaded hardware flows, which are compared with identified large software flows to determine which software flows to offload to hardware as well as which hardware flows to upload to software.

Algorithm 1 presents the pseudo code of the identification and replacement algorithm for large flows. Lines 1 calculates the number of the two-dimensional arrays, lines 2–4 describe the update process of the sketch data structure, lines 5–19 show the update process of the min-heap data structure, lines 20–24 illustrate the flow replacement process, and lines 25–28 demonstrate the update process of the identification window.

3.3 Event-Driven Bursty Flow Replacement

Elixir leverages an event-driven process to detect and offload bursty flows, *i.e.* when the software queue size exceeds a pre-defined threshold (K), one or more bursty flows in the software need to be offloaded to the hardware. It indicates that, if there is a bursty flow in the software but the queue size is smaller than the threshold, Elixir does not trigger event-driven replacement (instead, only periodic large flow replacement is carried on), so as to reduce the potential performance degradation caused by the frequent replacement. It is also worth noting that, Elixir can never promise there is no packet loss in the software forwarding system; instead, Elixir can only try its best to offload bursty flows that may cause packet loss. For an extreme example, if all the flows are bursty at the same time, it is impossible to offload all the flows immediately to avoid packet loss.

When triggering the bursty flow offloading process, hardware counters of the flows which are maintained in the hardware bursty area are polled to software. By a small identification window to compare the hardware and software flows, the corresponding flows are determined for replacement.

Setting the threshold of software queue size: Many modern practical software forwarding systems use high performance packet IO frameworks like DPDK [24] for packet processing. In these frameworks, the hardware and software communicate via a ring buffer and the hardware directly places packets into the ring buffer. When the software forwarding speed cannot catch up with the traffic receiving rate during the traffic burst period, packets will be lost at the NIC ring buffer and the queue size will become large. Apparently, a sudden increase of the ring buffer queue size is a clear signal of the existence of bursty flows.

Algorithm 2 The identification and replacement algorithm for bursty flows.

Input: The queue size threshold K.
1: **if** get_queue_size() $\geq K$ **then**
2:     $f_{bursty} \leftarrow$ get_hardware_bursty_flows()
3:     $c_{bursty} \leftarrow$ poll_flow_counters(f_{bursty})
4:     $f_{replaced} \leftarrow$ get_the_smallest_flow(f_{bursty}, c_{bursty})
5:     $f_{offloaded} \leftarrow$ get_offloaded_flow()
6:     **if** ($f_{offloaded}$.size > $f_{replaced}$.size) **then**
7:         replace_hardware_flow($f_{replaced}, f_{offloaded}$)
8:     **end if**
9: **end if**

One problem here is how to set the threshold K for triggering bursty flow offloading. We borrow the idea used in setting the headroom size in the PFC mechanism [1]. In PFC, the headroom should be large enough to store the packets received by a switch between the time when the PFC pause message is sent and the time when the PFC pause message takes effect. Similarly, when setting the threshold K in Elixir, a headroom should be reserved to store the packets received by the NIC between the time when bursty flow offloading is triggered and the time when the flows responsible for the burst are successfully offloaded. Following this principle, the threshold K is set to $K =$ queue_capacity $-(t_{identification} + t_{offloading}) \times$ NIC_speed, where $t_{identification}$ is the time required for identifying the bursty flows and $t_{offloading}$ is the time taken for completing the offloading process. With this queue-based detection mechanism, Elixir can detect bursty flows in a fast way while avoiding packet loss during offloading.

Setting the identification window size: When the queue size indicates the occurrence of bursty traffic, Elixir should identify the exact flow that causes the bursty traffic and replace it with a hardware flow that is no longer bursting. To achieve the purpose, Elixir not only associates each offloaded bursty flow with a hardware counter in the bursty flow area of the hardware as aforementioned, but also uses a counter to measure the accurate rate of each software-forwarded flow. In order to catch the flow burstiness, the identification windows of both software counters and hardware counters are small, *i.e.* they are reset every small time window. For the cloud gateway trace, we set the identification window for the counters to 1 second. As shown in Fig. 5(a), an identification window of 1 second can make bursty flows rank higher than stable flows with >60% probability.

It also indicates that, the additional cost of software counters is not high, since the total traffic volume forwarded by software is much smaller than that forwarded by hardware, and the software only needs to maintain a identification window of 1 second.

Identification and replacement algorithm for bursty flows: When traffic burst is detected in software, Elixir immediately polls the hardware counters to software. The process

is fast since the number of hardware counters in the bursty flow area is small. All the hardware counters and the software counters are sorted together to find the flows with the highest flow rates. A software flow will be offloaded if its rate is larger than that of a flow in the bursty flow area of hardware.

Algorithm 2 presents the pseudo code of identification and replacement algorithm for bursty flows in Elixir. Line 1 shows the queue-based bursty detection, lines 2–3 describe the process of polling the counters from the bursty flow area in hardware, and lines 4–9 demonstrate the replacement process of bursty flows.

It is worth noting that, the replacement processes of large flows and bursty flows are independent. Although their hardware flow tables are separated, the software flow table is shared between the two processes. Therefore, it may happen that both processes decide to offload a *bursty and large* software flow. In this case, the faster process will complete the offloading process, while the slower process will fail in flow offloading. It will not impact the correctness of flow replacement, and it does not matter which process finishes the flow offloading.

4   Implementation

We have implemented two Elixir prototypes. One is implemented on a server with an offloading-capable Mellanox ConnectX-5 NIC, and the other is implemented on a server connecting a Barefoot Wedge100BF-32X/65X P4 Switch. The switch supports both traffic sampling and hardware counters. The NIC only supports hardware counters currently, although Mellanox company plans to support the sampling functionality in the new release based on our knowledge. In our implementation, we direct the NIC traffic to a switch for sampling.

On the software side, we develop an Elixir library on top of DPDK (version 20.08), with ~3k lines of C codes. The library has three primary components, namely, software queue monitor, traffic analyzer and flow table manager. For offloading the flow tables to the NIC, Elixir manages the NIC hardware flow table with DPDK *rte_flow* API to support hairpin function, *i.e.* a flow bypasses the software when a hardware flow table entry is matched. For switch offloading, all the switch functions are implemented by programming the switch pipeline with PD API (Program Dependent API) [60].

Software queue monitor: Elixir leverages the software queue size as an indicator to offload software bursty flows. The queue size of the ring buffer can be calculated by polling several NIC pointers. We use the software consumer index pointer (rq_ci) and the hardware producer index pointer (rq_pi) to calculate the queue size as queue_capacity − (rq_ci − rq_pi). To achieve real-time bursty flow detection, Elixir reads the pointers upon each packet's arrival. For this reason, we implement the NIC pointer reading logic inside the DPDK packet receiving logic. Once a bursty flow is detected, the signal is passed to the flow table manager via a lockless queue.

Flow table manager: The flow table manager is responsible for managing the flow table entries in the hardware, *i.e.* offloading the flows selected by the traffic analyzer to hardware. For the NIC prototype, we leverage the *rte_flow_create* and *rte_flow_destroy* of the DPDK *rte_flow* API to insert and replace flow table entries in the NIC. For the switch prototype, we implement an RPC service in switch ONL (Open Network Linux) system, which modifies flow table entries via PD API when receiving offloading instructions from the flow table manager.

Traffic analyzer: Elixir implements the traffic analyzer component to identify the proper flows to offload. As aforementioned, the traffic analyzer identifies large flows through the sketch and min-heap data structures. In our implementation, it takes ~10ms to rebuild the heap after each replacement operation. For the NIC prototype, we leverage DPDK *rte_flow_query* to read the hardware flow counters from the NIC. For the switch prototype, the RPC service queries the flow counters and reports the results to the traffic analyzer.

5   Evaluation

In this section, we evaluate Elixir based on the prototypes we have developed. We first carry on a series of micro-benchmark experiments and then demonstrate the overall performance of Elixir.

5.1   Experimental Setup

Testbed: We build a testbed of 3 servers, namely, a packet sender, a packet receiver and a forwarder. The forwarder uses either the Elixir-NIC prototype (called the *Elixir-NIC forwarder*) or the Elixir-switch prototype (called the *Elixir-switch forwarder*), as described in Section 4. Each of the three servers is equipped with two 14-core Intel CPUs, 128GB RAM and 100GbE NICs. The OS of each server is ubuntu 16.04.5 LTS. For the *Elixir-NIC forwarder*, the NIC is Mellanox ConnectX-5 NIC with the firmware of 16.28.1002. For the *Elixir-switch forwarder*, the switch is Barefoot Wedge100BF-32X/65X P4 Switch, with one 4-core Intel CPU and 8GB RAM. We employ *dpdk_pkg_gen* version 20.08.0 together with DPDK version 20.05 to generate, forward and receive packets.

Comparison: We compare Elixir with three other solutions, namely, *Traffic-aware Flow Offloading (TFO)* [61], *Large Flow Predictor (LFP)* [15] and *Pure Software Forwarding (PSF)*. TFO and LFP represent the state-of-the-art hybrid flow table management solutions which only periodically replace large flows (without particular attention to bursty flows). TFO employs a sliding window to identify and replace large flows. LFP uses a learning method to predict and replace large flows. To keep consistency with the original papers [15, 61], we implement TFO in the Elixir-switch forwarder, while implement LFP in both the Elixir-switch

forwarder and the Elixir-NIC forwarder. PSF represents a baseline solution which forwards all the traffic by software, so it is only implemented in software.

Trace: We use the packet-level trace collected from the cloud gateway throughout this paper. The traffic shows a highly skewed pattern, namely, the top 10% largest flows comprise 91% of the total traffic. Besides, the packet sizes vary from 64 bytes to 1500 bytes. We use 10 minutes of the traffic trace for our experiments. When replaying the traffic trace, we control the average sending rate of the trace by *dpdk_pkt_gen*, so as to match different testing scenarios.

Elixir parameters: In the performance evaluation, we set the identification window of large flows to 30 seconds, the identification window of bursty flows to 1 second, the replacement window of large flows to 10 seconds, and the sampling rate to 20%. For Elixir, the large flow area in the hardware is set to contain at most 50k flows, while the bursty flow area in the hardware is set to contain at most 100 flows. For fair comparisons, in TFO and LFP, the hardware flow table is set to support 50.1k flows. The queue capacity of DPDK ring buffer is set to 64k packets. The queue threshold K is set to 28k packets for the Elixir-switch forwarder and 51k packets for the Elixir-NIC forwarder, according to the analysis in Section 3.3. For TFO, we use the default parameters in [61]. For LFP, we employ the J48 implementation [3] as the learning method and the first packet of each flow to predict the large flows, because this setting can achieve the best performance for LFP.

5.2 Micro-benchmarks

In the micro-benchmark experiments, we demonstrate that Elixir can identify the bursty flows timely and replace the identified bursty flows in a fast way. In addition, we show the advantage of Elixir over previous burst-oblivious replacement algorithms (TFO and LFP) with regard to bursty flow replacement.

Queue-based bursty flow detection: We purposely generate flows with the same average rate (∼5 Mpps) but different peak rates. When a flow sent from the sender begins to burst, we inject a special signal packet into the flow. The burst detection time is just the time interval between the time when the forwarder receives the signal packet and the time when the software queue size exceeds the threshold.

As shown in Fig. 11, the average detection time of bursty flows for Elixir-switch forwarder is 0.5∼6.0ms, while that for Elixir-NIC forwarder is 1.1∼11.0ms. The detection time is generally much lower than the peak rate duration of bursty flows, *i.e.* one or several seconds. The fast detection of the bursty flows gives us enough time to select and replace the bursty flows.

We also find out that the detection time of Elixir-switch forwarder is larger than that of Elixir-NIC forwarder, since it
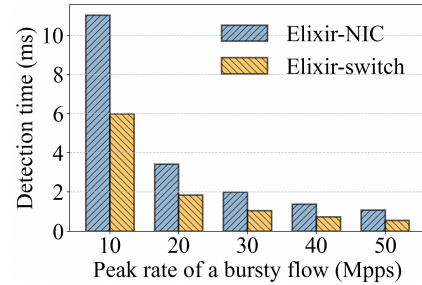
Figure 11: Detection time of bursty flows against different peak rates.
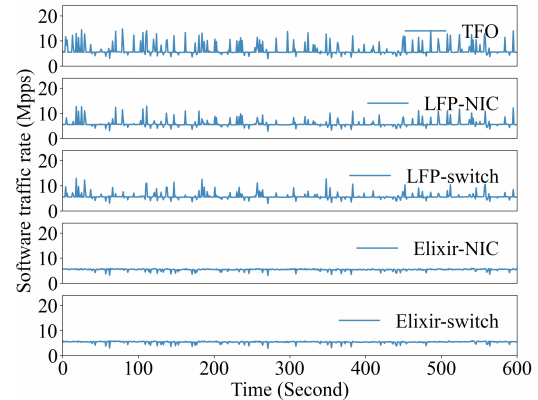
Figure 12: Software traffic rate of bursty flow replacement.

takes more time for a bursty flow to exceed the larger queue threshold in the Elixir-switch forwarder. Besides, the detection time decreases as the flow peak rate increases. This is because it takes less time for the software queue size to exceed the threshold when the flow rate is larger.

Bursty flow identification and replacement: Elixir identifies the flow that is responsible for the bursty traffic and exchange it with a hardware flow which is no longer bursting. In this experiment, we verify that the bursty flow identification and replacement algorithm can accurately select the flow responsible for the bursty traffic and replace it timely. We purposely generate 30-minute traffic, with 1000 bursty flows bursting at different times. The average rate of the traffic is ∼5 Mpps. The peak rates of the bursty flows follow a uniform distribution from 6 Mpps to 50 Mpps. The experimental results show that 984 out of the 1000 bursty flows can be successfully identified. Of the 16 bursty flows that are not identified, 13 flows have been offloaded to the hardware and do not cause the queue size to increase, and 3 flows are not identified due to the measurement errors.

We also measure the time between the detection of the bursty traffic and the replacement of the bursty flow. The results show that the Elixir-NIC forwarder requires ∼0.0666 ms and the Elixir-switch forwarder needs ∼0.184 ms to replace the bursty flow. Hence, Elixir can provide timely bursty flow replacement.

Software-forwarded traffic: To intuitively demonstrate the effectiveness of Elixir in offloading bursty traffic, we measure the rate of software-forwarded traffic under Elixir-

---

[3]The Java implementation of the C4.5 algorithm [56].

Table 1: Latency, queue size and packet loss rate of different flow table management solutions.

| Methods | Case 1: without packet loss | | | | Case 2: fixed sending rate | | | | Loss rate |
| | Latency (μs) | | Queue size (Packet) | | Latency (μs) | | Queue size (Packet) | | |
	Ave	Tail	Ave	Tail	Ave	Tail	Ave	Tail	
Elixir-NIC	63.6	102.5	663.1	1121.6	63.6	102.5	663.1	1121.6	0%
Elixir-switch	78.0	137.5	863.1	1620.1	78.0	137.5	863.1	1620.1	0%
TFO	70.6	3970.5	726.2	53283.1	1396.3	5055.9	18610.8	64231.8	25.6%
LFP-NIC	46.2	3948.1	613.8	52865.9	1338.5	5003.8	17752.3	64173.2	22.9%
LFP-switch	69.8	3966.2	653.3	53197.6	1361.2	5032.1	18113.4	64214.5	24.1%
PSF	88.1	4392.6	1041.8	59488.7	4857.5	5071.1	64220.4	64285.2	81.3%

NIC forwarder, Elixir-switch forwarder, TFO forwarder, LFP-NIC forwarder and LFP-switch forwarder respectively. As shown in Fig. 12, Elixir can successfully keep the software-forwarded traffic at a stable and low rate, while there are many software traffic bursts in other methods.

5.3 Overall Performance

We evaluate the overall performance of Elixir by comparing it with TFO, LFP and PSF on the cloud gateway trace.

CPU consumption with fixed forwarding capability: To evaluate the resource consumption of Elixir, we set a fixed average sending rate (56 Mpps) and measure the packet loss rate of these solutions in software against different numbers of CPU cores consumed. The least number of CPU cores which can forward the traffic without packet loss is the minimum required CPU resources of each solution. Note that in Elixir, TFO and LFP, CPU cores are required not only for software forwarding, but also for flow rate measurement and sorting. For fair comparisons, we count all the required CPU cores. Of course in PSF, all the CPU cores are used for forwarding only.

As shown in Fig. 13, in our experiment Elixir needs only 2 cores to forward the traffic without packet loss. In contrast, TFO and LFP need 4 forwarding cores to prevent packet loss, while PSF needs 11 cores. It indicates that, for forwarding the same traffic without packet loss, Elixir can save 81.8% CPU resources compared with PSF while saving 50.0% CPU resources compared with the state-of-the-art solutions. Although in our experiment the traffic sending rate is not very high, in operational networks, a network device, such as a cloud gateway or a NFV device, needs to process traffic with the speed of up to terabits per second. Dozens or even hundreds of servers with multiple CPU cores need to be deployed to support the high traffic rate. Hence in practice, Elixir can significantly save the CPU resources, which is extremely important in cloud environment.

Forwarding capability with fixed CPU consumption: We then measure the maximum forwarding rate of the hardware/software hybrid forwarder when there is no packet loss, by using only two CPU cores. For Elixir, TFO and LFP, the two cores are used for both packet forwarding and flow rate measurement; while for PSF, the two cores are both used for packet forwarding. As shown in Table 2, the maximum forwarding rate of Elixir (∼56 Mpps) is almost twice higher
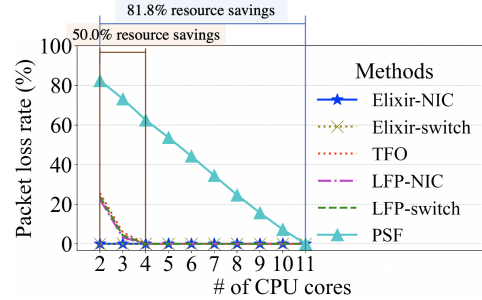
Figure 13: Packet loss rate against different numbers of CPU cores.

Table 2: Maximum forwarding rate of each solution by 2 CPU cores.

	Maximum forwarding rate (Mpps)
Elixir-NIC	56.2
Elixir-switch	56.1
TFO	26.4
LFP-NIC	28.3
LFP-switch	27.6
PSF	9.8

than that of *TFO* and *LFP* (∼27 Mpps). It indicates that, by burst-aware offloading, Elixir makes much higher utilization of the hardware forwarding capability. *PSF* has the lowest maximum forwarding rate (∼9.8 Mpps), since it does not leverage hardware offloading at all.

Latency, queue size and packet loss rate: We also compare Elixir with TFO, LFP and PSF by measuring the per-packet latency, software queue size and the packet loss rate if using two CPU cores at the forwarder. We test two different scenarios for these solutions. In the first case, for each solution the sender generates the highest-speed traffic if there is no packet loss at the forwarder. Specifically, the average sending rates of Elixir, TFO, LFP and PSF are 56.15 Mpps, 26.40 Mpps, 28.0Mpps and 9.80 Mpps, respectively. In the second case, a fixed sending rate of 56.15 Mpps is set for each solution, which is the maximum traffic rate for Elixir to avoid packet loss in the software.

Table 1 shows the results. In the first case, we find that Elixir has larger average latency and larger average queue size than TFO and LFP in the same hardware, since the traffic sending rate of Elixir is more than twice that of TFO and LFP. However, even in this case, the tail latency and tail queue size

of Elixir are orders-of-magnitude lower than that of TFO and LFP, due to the capability to timely detect and offload bursty flows. The gap is even amplified in the second case, where the same traffic sending rate is applied to all these solutions. Specifically, the tail latency of Elixir in the second case is ∼97.63% lower compared with TFO and ∼97.61% lower compared with LFP.

6 Related Works

Hardware/software hybrid flow table management: Previous works on hardware/software hybrid flow table management focus on two aspects, *i.e.* the application of the hybrid flow table and reasonable separation of the flow table between the hardware and software.

For the former aspect, most works focus on designing a network application based on the hybrid flow table [2, 31, 55]. TEA [31] designs a system architecture to enable programmable switches to query large virtual tables built on external DRAM. FlexGate [55] design a getaway to offload most of the traffic by the principle of traffic skewness. VPN Gateway [2] designs a getaway that can connect hundreds of enterprises to their thousands of VMs on the cloud.

The latter aspect is closer to Elixir, *i.e.* managing the hybrid flow table by splitting the flow table between the hardware and software [15, 29, 61]. TFO [61] leverages traffic skewness to split the flow table. CacheFlow [29] leverages traffic skewness and handles the dependencies between rules with overlapping patterns when splitting the flow table. LFP [15] introduces a machine-learning based approach to predict large flows with the first packet of each flow, and offload the predicted large flows. Hence, previous works on flow table splitting primarily focus on identifying and exchanging large flows between the hardware and software. Differently, Elixir pays special attention to bursty flows, and takes both large flows and bursty flows into consideration, which can make better utilization of the hardware resources compared to previous burst-oblivious solutions.

Traffic measurement to identify large flows and bursty flows: A line of previous works focus on traffic measurement to identify large flows [21, 23, 36, 41, 68, 69, 72] or bursty flows [9, 10].

For the large flow identification, they primarily adopt two kinds of approaches, which use exponentially decay and a sliding window separately. The works that employ a sliding window [6, 21, 66] record the flow sizes in a time window and update the flow sizes periodically. The works that adopt exponentially decay [11, 44, 62] apply a decay function which assigns weights to flow sizes based on their ages, and the history of the flow sizes is fading away. The flow sizes recorded by these approaches are used to identify large flows.

The identification of bursty flows is primarily based on a sliding window [9, 10], because exponentially decay is too coarse-grained to measure the fine-grained queue size. ConQuest [9] and Snappy [10] employ multiple snapshots (sketches) to record the flow sizes in a sliding window. They slide the window and reset the sketches periodically to measure the number of packets in the queue.

Elixir also identifies the large flows and bursty flows using an identification window with fixed size. Different from previous works, the large flows and bursty flows should be measured in hardware/software at the same time, which is achieved by sampling and hardware counters. Moreover, Elixir decouples the replacement window from the identification window due to system limitations, which is not considered by previous works.

NFV Optimization: Existing works on NFV optimization include *e.g.* elastic scaling [28, 57, 65], NFV chaining [51, 63], management [8, 51], hardware acceleration [18, 20, 22, 26, 27, 30, 34, 35, 40, 45–47, 53–55, 67], *etc.*. The works on hardware acceleration can be divided into two major directions: one is to offload all the network functions to the hardware with NICs [22, 30, 34, 35, 40, 46, 47, 54], switches [26, 27, 45, 47, 67] or GPUs [20], while the other is to only offload flow table entries to the hardware [2, 15, 29, 31, 55, 61]. Elixir can be applied in the latter case of hardware acceleration and make much higher utilization of the hardware capability than the state-of-the-art solutions.

Flow table compression and optimization: Many recent efforts have been taken on how to improve the utilization efficiency of flow table storage [4, 16, 32, 37, 38, 42, 59, 64, 71] and how to improve the lookup speed of the flow table [3, 14, 16, 32, 37, 38, 42, 58, 71]. They focus on the optimization of the organization of a single flow table. Differently, Elixir focuses on the management of both hardware and software flow tables to make better utilization of the hardware resources.

7 Conclusion

In this paper we design Elixir, a high-performance and low-cost approach to managing hardware/software hybrid flow tables. Compared with previous solutions which only offload large flows to the hardware, Elixir takes both large flows and bursty flows into consideration. Specifically, Elixir combines both sampling-based mechanism and counter-based mechanism for large flow and bursty flow rate measurement, separates the replacement processes of large flows and bursty flows, and decouples the identification window and the replacement window. By these techniques, Elixir not only considerably saves CPU resources required for software forwarding, but also achieves much lower tail forwarding latency.

Acknowledgement

References

[1] 802.1Qbb. 802.1Qbb - Priority-based Flow Control. *http://www.ieee802.org/1/pages/802.1bb.html*, 2008.

[2] Mina Tahmasbi Arashloo, Pavel Shirshov, Rohan Gandhi, Guohan Lu, Lihua Yuan, and Jennifer Rexford. A scalable vpn gateway for multi-tenant cloud services. *ACM SIGCOMM Computer Communication Review*, 48(1):49–55, 2018.

[3] Hirochika Asai and Yasuhiro Ohara. Poptrie: A compressed trie with population count for fast and scalable software ip routing table lookup. *ACM SIGCOMM Computer Communication Review*, 45(4):57–70, 2015.

[4] Subhasis Banerjee and Kalapriya Kannan. Tag-in-tag: Efficient flow table management in sdn switches. In *10th International Conference on Network and Service Management (CNSM) and Workshop*, pages 109–117. IEEE, 2014.

[5] Antonin Bas. Leveraging stratum and tofino fast refresh for software upgrades. *ONF CONNECT*, 2018.

[6] Ran Ben-Basat, Gil Einziger, Roy Friedman, and Yaron Kassner. Heavy hitters in streams and sliding windows. In *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9. IEEE, 2016.

[7] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[8] Anat Bremler-Barr, Yotam Harchol, and David Hay. Openbox: a software-defined framework for developing, deploying, and managing network functions. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 511–524, 2016.

[9] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, and Ori Rottenstreich. Catching the microburst culprits with snappy. In *Proceedings of the Afternoon Workshop on Self-Driving Networks*, pages 22–28, 2018.

[10] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, Ori Rottenstreich, Steven A Monetti, and Tzuu-Yi Wang. Fine-grained queue measurement in the data plane. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, pages 15–29, 2019.

[11] Graham Cormode, Flip Korn, and Srikanta Tirthapura. Exponentially decayed aggregates on data streams. In *2008 IEEE 24th International Conference on Data Engineering*, pages 1379–1381. IEEE, 2008.

[12] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. In *latin american symposium on theoretical informatics*, pages 29–38. Springer, 2004.

[13] Intel Corporation. Explore the Power of Intel Programmable Ethernet Switch Products. *https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch.html*, 2021.

[14] James Daly, Valerio Bruschi, Leonardo Linguaglossa, Salvatore Pontarelli, Dario Rossi, Jerome Tollet, Eric Torng, and Andrew Yourtchenko. Tuplemerge: Fast software packet processing for online packet classification. *IEEE/ACM transactions on networking*, 27(4):1417–1431, 2019.

[15] Raphael Durner and Wolfgang Kellerer. Network function offloading through classification of elephant flows. *IEEE Transactions on Network and Service Management*, 2020.

[16] Will Eatherton, George Varghese, and Zubin Dittia. Tree bitmap: hardware/software ip lookups with incremental updates. *ACM SIGCOMM Computer Communication Review*, 34(2):97–122, 2004.

[17] Dino Farinacci, T Li, S Hanks, D Meyer, and P Traina. Rfc2784: Generic routing encapsulation (gre), 2000.

[18] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure accelerated networking: Smartnics in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, 2018.

[19] Romain Fontugne, Patrice Abry, Kensuke Fukuda, Darryl Veitch, Kenjiro Cho, Pierre Borgnat, and Herwig Wendt. Scaling in internet traffic: a 14 year and 3 day longitudinal study, with multiscale analyses and random projections. *IEEE/ACM Transactions on Networking*, 25(4):2152–2165, 2017.

[20] Younghwan Go, Muhammad Asim Jamshed, Young-Gyoun Moon, Changho Hwang, and KyoungSoo Park. Apunet: Revitalizing gpu as packet processing accelerator. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 83–96, 2017.

[21] Xiangyang Gou, Long He, Yinda Zhang, Ke Wang, Xilai Liu, Tong Yang, Yi Wang, and Bin Cui. Sliding sketches: A framework using time zones for data stream processing in sliding windows. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1015–1025, 2020.

[22] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. Softnic: A software nic to augment hardware. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-155*, 2015.

[23] Qun Huang, Xin Jin, Patrick PC Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. Sketchvisor: Robust network measurement for software packet processing. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 113–126. ACM, 2017.

[24] DPDK Intel. Data plane development kit, 2014.

[25] Hao Jiang and Constantinos Dovrolis. Why is the internet traffic bursty in short time scales? In *Proceedings of the 2005 ACM SIGMETRICS international Conference on Measurement and Modeling of Computer Systems*, pages 241–252, 2005.

[26] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 35–49, 2018.

[27] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 121–136, 2017.

[28] Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. Stateless network functions: Breaking the tight coupling of state and processing. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 97–112, 2017.

[29] Naga Katta, Omid Alipourfard, Jennifer Rexford, and David Walker. Cacheflow: Dependency-aware rule-caching for software-defined networks. In *Proceedings of the Symposium on SDN Research*, pages 1–12, 2016.

[30] Antoine Kaufmann, SImon Peter, Naveen Kr Sharma, Thomas Anderson, and Arvind Krishnamurthy. High performance packet processing with flexnic. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 67–81, 2016.

[31] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. Tea: Enabling state-intensive network functions on programmable switches. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 90–106, 2020.

[32] Kirill Kogan, Sergey Nikolenko, Ori Rottenstreich, William Culhane, and Patrick Eugster. Sax-pac (scalable and expressive packet classification). In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 15–26, 2014.

[33] Georgios Y Lazarou, Julie Baca, Victor S Frost, and Joseph B Evans. Describing network traffic using the index of variability. *IEEE/ACM Transactions On Networking*, 17(5):1672–1683, 2009.

[34] Yanfang Le, Hyunseok Chang, Sarit Mukherjee, Limin Wang, Aditya Akella, Michael M Swift, and TV Lakshman. Uno: uniflying host and smart nic offload for flexible packet processing. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 506–519, 2017.

[35] Bojie Li, Kun Tan, Layong Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 1–14, 2016.

[36] Jizhou Li, Zikun Li, Yifei Xu, Shiqi Jiang, Tong Yang, Bin Cui, Yafei Dai, and Gong Zhang. Wavingsketch: An unbiased and generic sketch for finding top-k items in data streams. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1574–1584, 2020.

[37] Wenjun Li, Tong Yang, Ori Rottenstreich, Xianfeng Li, Gaogang Xie, Hui Li, Balajee Vamanan, Dagang Li, and Huiping Lin. Tuple space assisted packet classification with high performance on both search and update. *IEEE Journal on Selected Areas in Communications*, 2020.

[38] Eric Liang, Hang Zhu, Xin Jin, and Ion Stoica. Neural packet classification. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 256–269. 2019.

[39] Lingtong Liu, Yulong Shen, Yibo Yan, Tong Yang, Muhammad Shahzad, Bin Cui, and Gaogang Xie. Sf-sketch: A two-stage sketch for data streams. *IEEE Transactions on Parallel and Distributed Systems*, 31(10):2263–2276, 2020.

[40] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. E3: energy-efficient microservices on smartnic-accelerated servers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 363–378, 2019.

[41] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. Nitrosketch: Robust and general sketch-based monitoring in software switches. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 334–350. ACM, 2019.

[42] Zhi Liu, Xiang Wang, Baohua Yang, and Jun Li. Bitcuts: Towards fast packet classification for order-independent rules. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 339–340, 2015.

[43] Mallik Mahalingam, Dinesh G Dutt, Kenneth Duda, Puneet Agarwal, Lawrence Kreeger, T Sridhar, Mike Bursell, and Chris Wright. Virtual extensible local area network (vxlan): A framework for overlaying virtualized layer 2 networks over layer 3 networks. *RFC*, 7348:1–22, 2014.

[44] Sergiy Matusevych, Alex Smola, and Amr Ahmed. Hokusai-sketching streams in real time. *arXiv preprint arXiv:1210.4891*, 2012.

[45] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 15–28, 2017.

[46] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. Acceltcp: Accelerating network applications with stateful tcp offloading. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 77–92, 2020.

[47] Daniele Moro, Manuel Peuster, Holger Karl, and Antonio Capone. Fop4: Function offloading prototyping in heterogeneous and programmable network scenarios. In *2019 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 1–6. IEEE, 2019.

[48] Xena network. White paper: Is your network prepared for microbursts? *https://www.xenanetworks.com/wp-content/uploads/2019/11/Microburst_WP.pdf*, 2009.

[49] NoviFlow. NoviSwitch: SDN Switch, Switching Made Programmable. *https://noviflow.com/noviswitch*, 2021.

[50] NVIDIA. Mellanox ConnectX-5: Advanced Offload Capabilities for the Most Demanding Applications. *https://www.nvidia.com/en-us/networking/ethernet/connectx-5*, 2021.

[51] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: a framework for nfv applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 121–136, 2015.

[52] Konstantina Papagiannaki, Rene Cruz, and Christophe Diot. Network performance monitoring at small time scales. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pages 295–300, 2003.

[53] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The design and implementation of open vswitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 117–130, 2015.

[54] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: a programming system for nic-accelerated network applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 663–679, 2018.

[55] Kun Qian, Sai Ma, Mao Miao, Jianyuan Lu, Tong Zhang, Peilong Wang, Chenghao Sun, and Fengyuan Ren. Flexgate: High-performance heterogeneous gateway in data centers. In *Proceedings of the 3rd Asia-Pacific Workshop on Networking 2019*, pages 36–42. ACM, 2019.

[56] J Ross Quinlan. *C4. 5: programs for machine learning*. Elsevier, 2014.

[57] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. Split/merge: System support for elastic execution in virtual middleboxes. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 227–240, 2013.

[58] Alon Rashelbach, Ori Rottenstreich, and Mark Silberstein. A computational approach to packet classification. *arXiv preprint arXiv:2002.07584*, 2020.

[59] Gábor Rétvári, János Tapolcai, Attila Kőrösi, András Majdán, and Zalán Heszberger. Compressing ip forwarding tables: Towards entropy bounds and beyond. *ACM SIGCOMM Computer Communication Review*, 43(4):111–122, 2013.

[60] Yavuz Yetim Samar Abdi, Waqar Mohsin. P4 Program-Dependent Controller Interface for SDN Applications. *https://p4.org/assets/p4-ws-2017-p4-program-dependent-api-for-sdn-applications.pdf*, 2017.

[61] Nadi Sarrar, Steve Uhlig, Anja Feldmann, Rob Sherwood, and Xin Huang. Leveraging zipf's law for traffic offloading. *ACM SIGCOMM Computer Communication Review*, 42(1):16–22, 2012.

[62] Anshumali Shrivastava, Arnd Christian Konig, and Mikhail Bilenko. Time adaptive sketches (ada-sketches) for summarizing data streams. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1417–1432, 2016.

[63] Chen Sun, Jun Bi, Zhilong Zheng, Heng Yu, and Hongxin Hu. Nfp: Enabling network function parallelism in nfv. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 43–56, 2017.

[64] Zartash Afzal Uzmi, Markus Nebel, Ahsan Tariq, Sana Jawad, Ruichuan Chen, Aman Shaikh, Jia Wang, and Paul Francis. Smalta: practical and near-optimal fib aggregation. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*, pages 1–12, 2011.

[65] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. Elastic scaling of stateful network functions. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 299–312, 2018.

[66] Fangjia Xing, Fan Zhang, Xuxiang Tian, Wenyao Li, Hanhua Chen, and Hai Jin. Identifying the most recent heavy hitters in large-scale streams using block-wise counting. In *2017 IEEE 10th Conference on Service-Oriented Computing and Applications (SOCA)*, pages 239–244. IEEE, 2017.

[67] Ji Yang, Xiaowei Yang, Zhenyu Zhou, Xin Wu, Theophilus Benson, and Chengchen Hu. Focus: Function offloading from a controller to utilize switch power. In *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 199–205. IEEE, 2016.

[68] Tong Yang, Junzhi Gong, Haowei Zhang, Lei Zou, Lei Shi, and Xiaoming Li. Heavyguardian: Separate and guard hot items in data streams. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2584–2593. ACM, 2018.

[69] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 561–575. ACM, 2018.

[70] Tong Yang, Lingtong Liu, Yibo Yan, Muhammad Shahzad, Yulong Shen, Xiaoming Li, Bin Cui, and Gaogang Xie. Sf-sketch: A fast, accurate, and memory efficient data structure to store frequencies of data items. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 103–106. IEEE, 2017.

[71] Tong Yang, Gaogang Xie, YanBiao Li, Qiaobin Fu, Alex X Liu, Qi Li, and Laurent Mathy. Guarantee ip lookup performance with fib explosion. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 39–50, 2014.

[72] Tong Yang, Haowei Zhang, Jinyang Li, Junzhi Gong, Steve Uhlig, Shigang Chen, and Xiaoming Li. Heavy-keeper: An accurate algorithm for finding top-*k* elephant flows. *IEEE/ACM Transactions on Networking*, 27(5):1845–1858, 2019.

[73] Kok-Kiong Yap, Murtaza Motiwala, Jeremy Rahe, Steve Padgett, Matthew Holliman, Gary Baldus, Marcus Hines, Taeeun Kim, Ashok Narayanan, Ankur Jain, et al. Taking the edge off with espresso: Scale, reliability and programmability for global internet peering. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 432–445. ACM, 2017.

[74] Yang Zhou, Tong Yang, Jie Jiang, Bin Cui, Minlan Yu, Xiaoming Li, and Steve Uhlig. Cold filter: A meta-framework for faster and more accurate stream processing. In *Proceedings of the 2018 International Conference on Management of Data*, pages 741–756, 2018.

Gearbox: A Hierarchical Packet Scheduler for Approximate Weighted Fair Queuing

Peixuan Gao
New York University

Anthony Dalleggio
New York University

Yang Xu [*]
Fudan University

H. Jonathan Chao
New York University

Abstract

Bandwidth allocation and performance isolation are crucial to achieving network virtualization and guaranteeing service quality in data centers as well as other network systems. Weighted Fair Queuing (WFQ) can achieve customized bandwidth allocation and flow isolation; however, its implementation in large-scale high-speed network systems is very challenging due to the high complexity of the scheduling and the large number of queues required.

This paper proposes Gearbox, a scheduler primitive for next-generation programmable switches and smart NICs that practically approximates WFQ. Gearbox consists of a logical hierarchy of queuing levels, which accommodate a wide range of packet departure times using a relatively small number of FIFOs. Gearbox's enqueue and dequeue operations have $O(1)$ time complexity, which makes it suitable to cope with high-speed line rates. Gearbox provides its simplicity and performance advantages by allowing slight discrepancies in packet departure time from strict WFQ. We show that Gearbox's normalized departure time discrepancy is bounded and has a negligible impact on bandwidth allocation and flow completion time (FCT).

We implement Gearbox in NS2 and in VHDL, targeted to a Xilinx Alveo U250 card with an XCVU13P FPGA. The NS2 evaluation results show that Gearbox closely approximates WFQ and achieves weighted max-min fairness in bandwidth allocation as well as flow isolation. Gearbox provides FCT performance comparable to ideal WFQ. The Gearbox FPGA prototype runs at 350MHz and achieves full line rate for 100GbE with packets larger than 123 bytes. Gearbox consumes less than 1% of the FPGA's logic resources and less than 4% of its internal block memory.

1 Introduction

Bandwidth allocation and isolation are key to network virtualization in data centers and the performance of network systems (e.g., FCT, packet tail latency, and QoS guarantee). WFQ is an ideal packet scheduling scheme that can achieve bandwidth guaranties and performance isolation, as well as other objectives, such as minimizing average FCT and reducing tail packet latency.

However, it is very challenging to implement a WFQ packet scheduler in large-scale high-speed network systems (e.g., systems with millions of active flows and link capacities of hundreds of Gbps). Although several hardware schedulers have been proposed in the past to sort or sequence packets based on their departure times, most of them do not scale well with system size [8] [9] [10] [4] [17], except for a few cutting-edge SoC chips [11] [5]. Push-in First-out (PIFO) [27] [28] is a viable solution but it does not scale easily due to the large number of parallel rank comparisons. More recent works such as AFQ [24] and PCQ [25] are based on the idea of calendar queues [7] [31] and have been implemented on emerging programmable switches. However, to provide the packet serving order of an ideal scheduler, AFQ requires a large number of queues while PCQ suffers from additional memory accesses due to frequent packet re-circulation and packet migration between queues.

We observe that by allowing a slight packet departure order skew in the WFQ, we have the opportunity to greatly simplify its implementation. We define departure time discrepancy (DTD) as the difference between the system time when a packet is served and its departure time assigned by WFQ and normalized DTD as DTD normalized to the expected delay [1]. In fact, packets with different expected delays in ideal WFQ scheduling have different tolerances to such DTD. Packets with longer expected delays can tolerate larger DTD than those with shorter expected delays. By taking advantage of this observation, we propose a new packet scheduler, called Gearbox, that approximates WFQ with a bounded normalized packet DTD. Gearbox adopts the idea of calendar queues [7] [31] and places packets with different expected delays in different logical levels of the calendar queues upon their arrival. When implemented physically, the queues are in fact individual FIFOs arranged into independent groupings. Gearbox can accommodate a very large range of departure times while using a relatively small number of FIFOs. The simplicity of Gearbox's approach makes it suitable for implementation on next-gen programmable switches as well as smart NICs which typically include FPGAs [15]. Note that by changing the 'departure time' in WFQ to other priority ranks calculated by other scheduling schemes (e.g., re-

---

[*]Corresponding author

[1]The departure time refers to the virtual departure time in WFQ [14] [22] [23]. The system time refers to the virtual system time in WFQ. When all the admitted flows are active, the virtual system time runs as fast as the real time.

maining flow size in pFabric [2]), Gearbox can approximate other scheduling schemes as a programmable scheduler. We summarize the contributions of this paper as follows:

- **Provide a close approximation of WFQ with bounded normalized DTD for packets.** Gearbox schedules packets with different expected delays using calendar queues with different granularities. Such tiered granularity allows Gearbox to closely approximate ideal WFQ with a relatively small number of FIFOs while guaranteeing a bounded normalized DTD.

- **Offer scalability with simple implementation.** Gearbox is implemented using queues (FIFOs) from a flat array that are arranged into logical levels. This non-hierarchical FIFO-based physical implementation allows Gearbox to admit a large number of packets and does not presume the existence of queue hierarchy in next-gen programmable switches [24] [25].

- **Implement Gearbox in NS2 with extensive packet-based evaluations.** We implement Gearbox in NS2 [21] and perform extensive packet-based evaluations. The results show that Gearbox closely approximates ideal WFQ and achieves a good weighted max-min fairness with per-flow isolation even in a short time scale. Our simulations based on a fat-tree topology show that Gearbox has an FCT performance closely matching the ideal WFQ using a PIFO.

- **Implement Gearbox in VHDL targeting an FPGA.** We implement a Gearbox VHDL prototype and target a Xilinx Alveo U250 FPGA card with a mid-speed grade XCVU13P FPGA. The Gearbox prototype runs at 350MHz and reaches full 100GbE line rate with packets larger than 123 bytes[2]. Based on the implementation report, Gearbox uses less than 1% of the target FPGA logic resources and less than 4% of its block random access memories (BRAM) as detailed further in Section 4.

The rest of the paper is organized as follows. Section 2 introduces the background and motivation of our work. Section 3 presents the detailed mechanism of the Gearbox scheduler and its extensions. Section 4 presents NS2 evaluation and the Gearbox hardware prototype. We discuss related works in Section 5 and conclude the paper in Section 6.

2 Background and Motivation

2.1 Weighted Fair Queuing

Bandwidth allocation and isolation are critical in data centers as well as other network systems [13] [18] [30] [3] [19].

These attributes are also essential to network performance such as max-min fairness, FCT, and tail latency [20].

WFQ [14] [22] [23] is the most effective algorithm to allocate bandwidth among flows and provide per-flow isolation. WFQ assigns a departure time to each packet and guarantees bandwidth allocation by scheduling packets in ascending order of their departure time[3]. For each packet scheduled by WFQ, its expected delay is its packet size divided by its assigned rate provided that the flow's traffic has been shaped. The expected delay is equal to the difference between the assigned departure time and the system time when the packet arrives.

2.2 Challenges of a WFQ Packet Scheduler

The major challenge of implementing WFQ on high-speed switches is the limited time to process each packet. For 64-byte packets, a scheduler for a 100GbE link has a processing time of only 6.72 ns[4]. In addition, sorting packets according to their departure time usually has a time complexity of $O(logN)$ [12], where N is the number of total packets or flows and could be on the order of several thousands. Maintaining a sorted list of packets in the limited packet processing time on high-speed links is very challenging.

An ASIC-based hardware sorter called Sequencer [8] [9] [10] inserts each arriving packet into a proper position in a queue according to its departure time. However, the Sequencer is not very scalable due to its high power consumption and chip area cost to support the parallel comparison of every arriving packet's departure time to all others' in the queue. The limited scalability of Sequencer precludes it from being used in a typical shared-memory switch in a data center [6] (e.g., with buffer size of ∼60K packets). The pipeline heap (P-heap) [4] [17] implements a WFQ scheduler with better scalability, but it is required for each output port, which results in significant chip area consumption [27], making a P-heap based WFQ scheduler on commodity switches less practical. Recent work on Pushed-in First-out (PIFO) [27] [28] performs parallel packet rank comparisons (similar to above-mentioned Scheduler), which limits its applicability in large high-speed switches.

A calendar queue, introduced by Randy Brown [7] and used in Timer Wheels [31] and used in Approximate Fair Queuing (AFQ) [24] and Programmable Calendar Queues (PCQ) [25], is scalable due to its implementation simplicity (only uses FIFOs). As a trade-off, calendar queues relax the packet sorting order compared to an ideal WFQ scheduler. Unlike in ideal WFQ, a calendar queue only sorts packets approximately in the ascending order of their departure times

---

[3]In a WFQ scheduler, the packet with the smallest departure time leaves the scheduler first.

[4]It is possible to reduce the scheduling rate by using input queuing to combine small packets into a single larger packet. E.g., two 64-byte packets belonging to the same flow can be scheduled as a single 128-byte packet.

---

[2]The VHDL design runs at 350 MHz in the target FPGA and performs enqueues and dequeues every 4 clock cycles, sustaining a packet rate of 87.5 Mpkts/sec. For Ethernet with a Preamble of 8 bytes and IFG of 12 bytes, the line bit rate for 123-byte packets is: $87.5M \times (8+123+12) \times 8 \approx 100Gb/s$.

since its accuracy is limited by the granularity of the sorting buckets. Moreover, calendar queues also suffer from calendar range overflow when the available number of queues cannot accommodate the wide range of departure times. To address the range overflow problem, modifications to calendar queues have been proposed to accept packets with larger departure times instead of dropping them [7][25]. These modifications consist of recirculating out-of-range packets to the end of the queues to delay them until they reach their departure times. When implemented on high-speed switches, calendar queue packet schedulers with packet recirculation impose additional demands on memory bandwidth due to the accesses for both incoming and recirculated packets. We discuss this further in Section 2.4.

2.3 Granularity of the Scheduler and Departure Time Discrepancy

The calendar queue scheduler trades some packet serving order accuracy for simplicity and scalability. Rather than sorting packets perfectly in the ascending order of their departure times, the calendar queue scheduler only sorts packets approximately compared to ideal WFQ. This approximation is due to the FIFO-based structure of the calendar queue [7]: a calendar queue only sorts packets by placing them into different FIFOs or buckets. Each FIFO in a calendar queue cannot differentiate the departure times of the packets in it. Here we quantify the scheduling precision of a calendar queue scheduler as the 'granularity of the scheduler'.

Granularity of the scheduler: the minimal departure time difference that a scheduler can discriminate, noted as g.

Consider the example in Figure 1. For the scheduler in Figure 1(a), each queue represents one virtual time unit and the scheduler can discriminate between the departure times with a difference of 1, which means it has a granularity $g = 1$. The scheduler in Figure 1(b) applies a coarser granularity $g = 10$. This scheduler can only discriminate between packet departure times based on the tens digit of their departure time.

With tiered granularities, calendar queue schedulers sort packets into different approximate orders when compared with ideal WFQ. We formally define the concept of 'departure time discrepancy (DTD)' to quantify the difference between the approximate serving order and that of ideal WFQ.

Departure time discrepancy (DTD): the difference between the system time when a packet leaves the scheduler and its departure time as scheduled by WFQ. We denote the kth packet in flow i as $P_{(i,k)}$ and its DTD as $d_{(i,k)}$.

$$d_{(i,k)} = \begin{cases} D_{(i,k)} - F_{(i,k)} & , D_{(i,k)} > F_{(i,k)} \\ 0 & , \text{otherwise} \end{cases} \quad (1)$$

Here $F_{(i,k)}$ is the ideal departure time as calculated by the

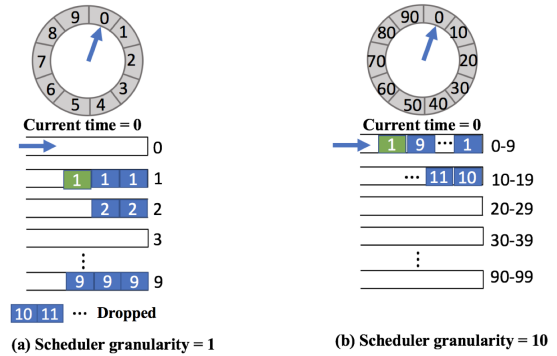(a) Scheduler granularity = 1   (b) Scheduler granularity = 10

Figure 1: Different granularities

WFQ scheduler for $P_{(i,k)}$ and $D_{(i,k)}$ is the actual departure time of packet $P_{(i,k)}$.

DTD shows the difference of the packet serving order between a calendar queue scheduler and an ideal WFQ. It indicates how well a calendar queue scheduler approximates an ideal WFQ from a packet's perspective. A small DTD indicates that the scheduler approximates an ideal WFQ closely and a large DTD, that a packet may experience a delay larger than its expected delay, which further leads to increased FCT or other consequences.

For a calendar-queue-based scheduler with a limited number of queues and fixed granularity [24] [25], different levels of granularity have pros and cons. Schedulers with finer granularity provide smaller DTDs. However, they only accommodate a narrow range of departure times and are therefore more prone to calendar range overflow. A rotating calendar scheduler consisting of M queues covers only $M * g$ future virtual time units. Packets with departure times beyond this bound will be dropped due to calendar range overflow, as shown in Figure 1(a). With a small number of available queues, we can consider the scheduler as having a shallow buffer, which is easy to overflow. For services requiring a certain amount of burstiness tolerance, this shallow buffer could lead to frequent packet drops. Furthermore, flows assigned relatively low bandwidths, which lead to large departure times, could experience excessive packet drops. This makes such fine-grained schedulers unsuitable to handle large variations in weighted bandwidth allocation. Schedulers with a coarser granularity alleviate the calendar range overflow issue by covering a wider range of departure times in the future. However, one obvious downside of using a coarse granularity is larger DTDs. Consequently, packets in the same queue may not be scheduled according to the order of their departure times. For example, in Figure 1(b) the green packet departs after packets with larger departure times in the coarse-grained scheduler.

As shown in the examples in Figure 1, scheduler granularity leads to a trade-off between fewer calendar range overflows and smaller DTDs.

2.4 Normalized DTD

How do we determine an appropriate granularity for the scheduler? One could argue that it is always better to have a scheduler with finer granularity. But is it necessary to achieve such a small DTD for all packets?

If we need to maintain a fine granularity while covering a wide departure time range, the total number of queues required would become very large. This design approach conflicts with the goal of a simple implementation. The time complexity of managing thousands of queues is very high and most switches may not have the required number of queues.

Another approach for providing fine scheduler granularity while preventing calendar range overflow is packet recirculation. The classic calendar queue [7] and Timer Wheel [31] place all packets into their queues. Packets scheduled too far in the future are simply enqueued in a queue roughly determined by the departure time modulo the total number of queues, and recirculated before their departure times become due. This will lead to additional memory bandwidth consumption, where memory bandwidth is very often the limiting factor in a networking equipment [32] [16]. Excessive packet recirculation can use up valuable shared memory bandwidth and lead to a significant drop in switch throughput. A recent work, PCQ [25], implements an alternative packet recirculation scheme. PCQ arranges calendar queues in multiple levels with different granularities. When PCQ finishes serving all the packets in the lower level, it recirculates and deposits all packets from a head queue in the higher level to appropriate queues in the lower level. Such a packet recirculation scheme can still lead to throughput reduction on high-speed switches.

Based on the above analysis, providing fine granularity with a wide range of packet departure times is resource intensive. But is it really necessary? We observe that packets have different tolerances to DTDs depending on their expected delays upon arrival as mentioned in Section 2.1. Packets with a smaller expected delay usually expect to be served shortly and are sensitive to small differences in the scheduling order. For these packets, a fine-grained scheduler is necessary to guarantee a small DTD. However, packets with larger departure times upon arrival can tolerate larger DTDs. These packets usually belong to flows with low assigned bandwidths: WFQ assigns these packets with large departure times to achieve bandwidth weighted max-min fairness in the long run. For these packets, their large expected delay makes it unnecessary to schedule them with fine granularity.

According to the above analysis, it is more appropriate to consider a packet's DTD based on its tolerance to it. We therefore normalize a packet's DTD to its expected delay.

Normalized DTD: the DTD normalized to the packet's expected delay, noted as $d_{n(i,k)}$.
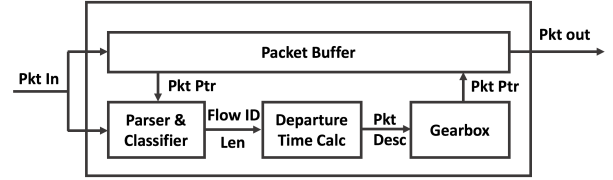

Figure 2: Gearbox System-level Application

$$d_{n(i,k)} = \begin{cases} \frac{D_{(i,k)} - F_{(i,k)}}{F_{(i,k)} - A_{(i,k)}} & , D_{(i,k)} > F_{(i,k)} \\ 0 & , \text{otherwise} \end{cases} \quad (2)$$

Here $A_{(i,k)}$ is the system time t_s when packet $P_{(i,k)}$ arrives at the scheduler, $F_{(i,k)}$ is the ideal departure time for packet $P_{(i,k)}$ determined by the WFQ scheduler, and $D_{(i,k)}$ is the actual departure time of packet $P_{(i,k)}$.

For example, say that packet $P_{(A,1)}$ and packet $P_{(B,1)}$ arrive at the scheduler when system time $t_s = 10$. Packet $P_{(A,1)}$ has an ideal departure time $F_{(A,1)} = 11$ and packet $P_{(B,1)}$ has an ideal departure time $F_{(B,1)} = 91$. Assume the scheduler applies a coarse granularity $g = 10$. Due to the coarse granularity, packet $P_{(A,1)}$ leaves the scheduler at $t_s = 19$ and packet $P_{(B,1)}$ leaves the scheduler at $t_s = 99$. Both of the packets have a DTD of 8. Although the two packets have the same DTD, they have different tolerances to it. The expected delay of packet $P_{(A,1)}$ and $P_{(B,1)}$ is $11 - 10 = 1$ and $91 - 10 = 81$ respectively. When we evaluate their normalized DTD, we have $d_{n(A,1)} = 8/1 = 8$ and $d_{n(B,1)} = 8/81 \approx 0.1$. This means packet $P_{(A,1)}$ is experiencing a delay that is 8 times its expected delay while packet $P_{(B,1)}$ has a delay only 1.1 times its expected delay. This indicates such granularity is appropriate for packet $P_{(B,1)}$ but is too coarse for packet $P_{(A,1)}$. This example shows that it is difficult to find an appropriate fixed granularity to schedule packets with different expected delays while satisfying DTD bounds.

3 Gearbox: Hierarchical Packet Scheduler

3.1 Basic Idea of Gearbox

Based on our analysis in Section 2.4, it is more appropriate to guarantee different DTD bounds for packets with different expected delays. In this case, we need a scheduler with flexible granularity to serve different packets. Thus, the scheduler must guarantee a low normalized DTD while keeping the implementation simple. We introduce Gearbox, a hierarchical packet scheduler that closely approximates WFQ and is simple to implement.

Figure 2 shows a simplified system-level application of Gearbox. Incoming packets are stored in the Packet Buffer. The packet header is sent to the packet Parser & Classifier to identify the flow (e.g., using 5-tuple classification) and determine the packet length. The Departure Time Calculator com-
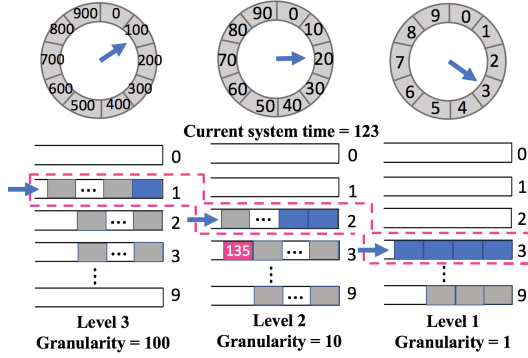
Figure 3: Gearbox: Hierarchical FIFO-based scheduler

Table 1: TERMS AND NOTATIONS

Notation	Description
$P_{(i,k)}$	k^{th} packet in flow i
$F_{(i,k)}$	Departure time of packet $P_{(i,k)}$ assigned by WFQ
$A_{(i,k)}$	System time when packet $P_{(i,k)}$ arrives at the scheduler
$D_{(i,k)}$	System time when packet $P_{(i,k)}$ leaves the scheduler
$d_{(i,k)}$	Departure time discrepancy (DTD) of packet $P_{(i,k)}$
$d_n(i,k)$	Normalized DTD of packet $P_{(i,k)}$
t_s	Current system virtual time
L	Total levels of the scheduler
M_l	Total number of FIFOs at level l
g_l	Granularity of level l
$Q_{(l,f)}$	FIFO f at level l

putes the packet's departure time and forwards it along with the packet descriptor (length, departure time, and enqueue Packet Pointer from Packet Buffer) to Gearbox to enqueue the packet. Upon request from a controller (not shown) Gearbox dequeues stored packet descriptors according to a calendar order described in detail below and forwards the descriptors to the Packet Buffer that outputs the packets. We further present the smart NIC use case and multi-pipeline use case of Gearbox in Appendix B.

Gearbox arranges FIFO-based calendar queues into 'logical levels' with different granularities, achieving the benefits of calendar queue schedulers with different granularities to serve packets with different expected delays. Lower logical levels with finer granularity provide smaller DTD for packets with departure times that are close to current system time upon arrival. Higher logical levels with coarser granularity serve packets with large departure times and cover a wider departure time range, reducing packet loss due to calendar range overflow. By providing tiered scheduling granularity, Gearbox guarantees a low normalized DTD for all packets with a relatively small number of queues. Moreover, Gearbox eliminates packet recirculation by directly serving packets at all levels based on our 'Compound FIFO' concept introduced in Section 3.2. This eliminates the additional memory accesses due to recirculation and allows Gearbox to achieve a high packet processing rate efficiently.

Figure 3 shows an example of the Gearbox hierarchical scheduler. Consider a scheduler with 30 available queues. Gearbox arranges them into 3 logical levels with different granularities, each logical level containing 10 queues. Level 1 has the finest granularity $g_1 = 1$, level 2 has a coarser granularity, $g_2 = 10$ and level 3 has the coarsest granularity, $g_3 = 100$. In this example, packets with a departure time within 10 virtual time units in the future upon arrival are enqueued in level 1 and are scheduled with the granularity of $g = 1$. Packets with a departure time larger than 10 but smaller than 100 virtual time units in the future upon arrival are enqueued in level 2. Other packets with departure time between 100 and 1000 virtual time units in the future upon

arrival are enqueued in level 3.

Note that the 'levels' in Gearbox are logical concepts. When implemented physically, they are in fact individual queues (FIFOs) arranged into independent groupings, which means Gearbox only requires a single level of queues in the hardware. We'll further discuss this in Sections 3.2 and 3.3. As a result, Gearbox can be implemented on devices that do not support hierarchy.

In section 3.4, we prove that Gearbox guarantees a low normalized DTD for all packets. Our evaluation shows that Gearbox closely approximates ideal WFQ from a user's point of view. Gearbox can be thought of as a clock. The lower logical level with finer granularity is like the second hand, serving packets in units of seconds. The higher logical levels are like the minute hand or the hour hand, admitting more packets with a larger departure time and serving them with a coarser granularity. Each logical level of the scheduler cooperates to schedule packets with different granularities, which makes the scheduler just like a gearbox shifting between different gears.

To better illustrate the detailed schemes in Gearbox, we summarize the related concepts and notations in Table 1. We further generalize the multi-level architecture of the Gearbox scheduler. The scheduler contains L logical levels of calendar queues, each one with a different granularity g_l. Each logical level of the scheduler contains M_l FIFOs and covers a departure time range of $[t_s, t_s + (M_l * g_l))$.

3.2 Compound FIFO

Researchers have tried to arrange the calendar queues in hierarchical structures, however, their implementations are not suitable for the ultra-high-speed switches due to packet recirculation and the short packet processing time.

To eliminate packet re-circulation, Gearbox applies the concept of a 'Compound FIFO' to directly schedule packets in different levels efficiently. As Figure 3 shows, the compound FIFO consists of the current serving FIFO in each level. In the example, the compound FIFO consists of $FIFO1$ at level 3, $FIFO2$ at level 2 and $FIFO3$ at level 1,

representing the current system time $t_s = 123$. Note that all the FIFOs in the compound FIFO cover the current system time t_s, which means that the individual FIFOs that make up the compound FIFO possibly contain packets that need to be served at the current system time.

Gearbox serves packets in the compound FIFO using a factor that is inversely proportional to the granularity of the level that each queue belongs to. Gearbox serves packets from the lowest-level queue to the highest-level queue in the compound FIFO. Since the lowest-level has the finest granularity, the departure time of all the packets in this queue equals the current system time t_s. Gearbox serves all the packets in the queue in the lowest level. After draining packets in the lowest-level queue, Gearbox starts to serve the queue at the next higher level. At each of the higher levels, we serve a number of bytes (rounded up to a whole number of packets) that is inversely proportional to the level's granularity g_l [5]. Gearbox finishes serving the compound FIFO when it finishes serving all the queues within it according to their levels. After serving the compound FIFO, Gearbox updates current system time t_s to the next non-empty compound FIFO.

As an illustration of serving packets from the different levels, the FIFOs in the pink box in Figure 3 are dequeued as follows. Gearbox first drains all the packets in $FIFO3$ at level 1. After that, since level 2 has the granularity of $g_2 = 10$ and $FIFO2$ at level 2 covers the time range from 120 to 129, Gearbox serves $1/10$ packets in this queue. Likewise, Gearbox serves $1/100$ of the packets in $FIFO1$ at level 3. After serving the correct proportion of packets in the queue at each level, Gearbox increases its current system time t_s by 1 and updates the compound FIFO based on the new system time $t_s = 124$.

The key contribution of the 'Compound FIFO' is freeing the scheduler from packet recirculation. By directly serving queues in different levels using a factor that inversely proportional to their granularity, Gearbox eliminates the need to recirculate packets. This means Gearbox's dequeue process is as simple as popping from a FIFO. This allows Gearbox to easily achieve a high packet processing rates on core switches. The trade-off is increased DTD in the higher levels.

3.3 Enqueue and Dequeue Processes

Enqueue Process The Gearbox packet enqueue process is straightforward. To enqueue a packet, Gearbox needs to determine the correct level and the destination FIFO for the arriving packet. Gearbox first finds the level to enqueue using the difference between the current system time t_s and the departure time of the packet $F_{(i,k)}$. The packet will enqueue

---

---

the lowest possible level that covers this interval. After finding the enqueue level, Gearbox simply divides the interval mentioned above by granularity g_l at this level to find the corresponding FIFO to enqueue.

Figure 3 shows an example of Gearbox's enqueue process. In the example, Gearbox has 3 levels covering the departure time range $[t_s, t_s + 9]$, $[t_s, t_s + 99]$, $[t_s, t_s + 999]$, respectively. The pink packet with a departure time of 135, arrives at the scheduler when $t_s = 123$. Upon the packet's arrival, the scheduler calculates the interval $F_{(i,k)} - t_s = 12$. This interval falls into the virtual time range covered by level 2. When we divide the interval 12 by the granularity g_2, we have $\lfloor 12/10 \rfloor = 1$, which indicates that this packet is enqueued in the FIFO next to the current serving FIFO in this level. The enqueue process is summarized in Algorithm 1.

Algorithm 1 Enqueue Process

1: **function** ENQUEUE PACKET($P_{(i,k)}$)
2:     **for** level l from 1 to L **do**
3:         **if** $\lceil (F_{(i,k)} - t_s)/g_l \rceil \le M_l$ **then**
4:             $f = \lfloor (F_{(i,k)} - t_s)/g_l \rfloor$
5:             $P_{(i,k)}$ enqueue f^{th} FIFO following the current serving FIFO
6:             **return**
7:     Drop packet $P_{(i,k)}$

Dequeue Process As we have introduced the compound FIFO in *Section 3.2*, the dequeue process of Gearbox was described above as serving the compound FIFO. We generalize Gearbox's dequeue process with the pseudo-code in Algorithm 2, where f_l is the FIFO of level l in the compound FIFO

Algorithm 2 Dequeue Process

1: **function** DEQUEUE PROCESS
2:     **for** level l from 1 to L **do**
3:         **if** $Q_{(l,f_l)}$ is not empty **then**
4:             dequeue $Q_{(l,f_l)}$ up to $Size(Q_{(l,f_l)})/g_l$

3.4 Normalized DTD Analysis

We previously defined the normalized DTD $d_{n(i,k)}$ as the DTD normalized to the packet's expected delay in equation (2). Now we need to determine its bound in Gearbox. Based on the above expression, the maximum value of $d_{n(i,k)}$ occurs when DTD is at the maximum value and the expected delay is at the minimum value. For a packet at level l, the maximum DTD equals the level's granularity g_l. Then we have:

$$\max\{D_{(i,k)} - F_{(i,k)}\} = g_l \qquad (3)$$

We need to find the minimal expected packet delay possible at level l. Based on the enqueue process described in section 3.3, packets enqueue at different levels based on their expected delay. Only packets with an expected delay between g_l and $g_l * M_l$ will enqueue at level l. Therefore, we have minimal expected delay:

$$\min\{F_{(i,k)} - A_{(i,k)}\} = g_l \qquad (4)$$

From equation (3) and (4), we have maximum delay:

$$\max\{D_{(i,k)} - A_{(i,k)}\} = 2g_l \qquad (5)$$

Then from (4) and (5) we have maximum normalized DTD:

$$\max\{d_{n(i,k)}\} = \max\{\frac{D_{(i,k)} - F_{(i,k)}}{F_{(i,k)} - A_{(i,k)}}\} = \frac{2g_l - g_l}{g_l} = 1 \qquad (6)$$

The maximum normalized DTD has an upper bound of 1, which means, in the worst case, that a packet scheduled by Gearbox will have a maximum delay that is twice its expected delay in WFQ. In other words, a packet that expects t microseconds delay in a WFQ scheduler will experience at most $2t$ microseconds delay in Gearbox in the worst case[6]. We evaluate the normalized DTD in actual time using simulations and provide our evaluation results in *Section 4*.

3.5 The Packet Out-of-order Issue

Packets in the hierarchical calendar queue scheduler may experience a packet out-of-order issue. According to the architecture of Gearbox, FIFOs at different levels might cover an overlapping virtual time range. As a result, a packet $P_{(i,k)}$ with a departure time $F_{(i,k)}$ may enqueue at any level of the scheduler, depending on its arrival time $A_{(i,k)}$. Similarly, a subsequent packet $P_{(i,k+1)}$ from the same flow may enqueue at a different level. When two packets from the same flow enqueue in different levels in the scheduler, the relative order of the packets is not deterministic. It is possible that packet $P_{(i,k)}$ leave later than the subsequent packet $P_{(i,k+1)}$ if it is enqueued at a different level in the scheduler, causing the packets to be dequeued out of order.

Figure 4 shows an example of the packet out-of-order issue. In the figure, packets $A1$ and $A2$ arrive at the scheduler when $t_s = 0$. Since both packets have a departure time exceeding $t_s + 9$, they cannot be enqueued at level 1 and are thus enqueued in $FIFO1$ at level 2. Later, the system time t_s updates to $t_s = 10$. A new packet $A3$ from the same flow arrives at the scheduler with a departure time of $F_{(A,3)} = 12$. At this moment, $F_{(A,3)} - t_s = 2 < 10$ and the scheduler places $A3$ in $FIFO2$ at level 1. At this point, packets from flow A

---

[6]The delay discrepancy may be smaller than 2x if all the flows are not active and the packet has an earlier opportunity to be dequeued
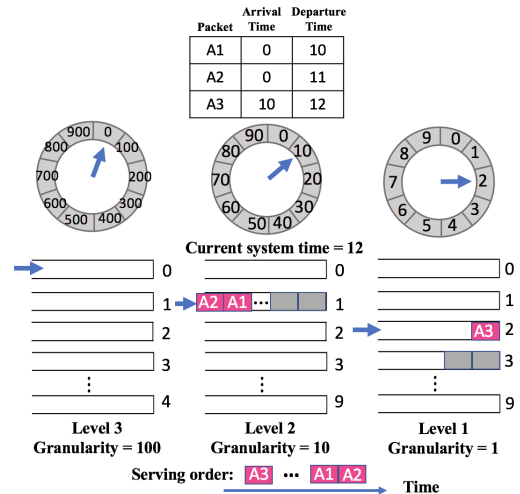

Figure 4: The packet out-of-order issue

are placed at different levels. As the scheduler starts to serve packet $A3$ when $t_s = 12$, packets $A1$ and $A2$ are still queued up at the tail of $FIFO1$ at level 2. In this case, packet $A3$ leaves before packets $A1$ and $A2$.

3.6 Solution to the Packet Out-of-order Issue

We introduce two modifications to eliminate the uncertainty in the packet serving order. First, we do not 'wrap around' FIFOs in the same level. This means that after the scheduler drains a FIFO and starts to serve the next one, the drained FIFO does not enqueue any packets until the scheduler finishes serving all the packets in the entire level. Second, we track the 'last packet enqueued level' for each flow, noted as L_i, where i is the flow id. When a new packet $P_{(i,k)}$ arrives at the scheduler, it is enqueued in a level equal to or higher than level L_i.

Let's consider the same example in Figure 4 after applying the solution. When packet $A1$ and $A2$ enqueue level 2 of the scheduler, Gearbox marks flow A's last packet enqueue level as $L_A = 2$. When packet $A3$ arrives at the scheduler, although $F_{(A,3)} - t_s = 2 < 10$, the scheduler will only place packet $A3$ into a level equal or larger than $L_A = 2$. In this case, Gearbox places packet $A3$ in $FIFO1$ at level 2 right after packet $A1$ and $A2$. As a result, it is scheduled after its preceding packets $A1$ and $A2$. Consequently, these modifications eliminate the packet out-of-order issue.

The modifications mentioned above may lead to side effects that could potentially increase the DTD of packets. The flows with input rates higher than their allocated rates will enqueue and stay in a higher level, where they will suffer from a coarser granularity and higher DTD. We further introduce the 'Step-down FIFO', an extension of Gearbox to solve these side effects. We present the details of this extension in Appendix A.
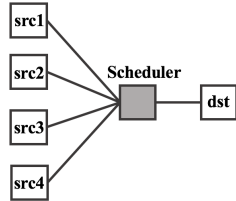
Figure 5: Single-node topology

Figure 6: Three-level fat-tree topology

Figure 7: Web-search flow size distribution

4 Implementation and Evaluation

In this section, we describe the design and implementation of Gearbox in NS2, a packet-based simulator [21], and in VHDL as a hardware prototype (targeted to Xilinx's Alveo U250 card[34]), along with the extensive simulations to evaluate the performance of Gearbox. The two implementations and their associated verification environments enabled us to explore different aspects of Gearbox including its performance in networks, the performance of the hardware prototype, and the hardware resource utilization.

4.1 Packet-based Evaluation

To evaluate Gearbox's performance in a large-scale network topology with real-world data traffic, we implemented Gearbox in NS2 [21] and conducted extensive packet-based simulations.

4.1.1 Evaluation Setup

Network topology We set up two different network topologies in NS2: (1) a single-node star topology for bandwidth allocation and fairness evaluation, and (2) a fat-tree topology to evaluate FCT and normalized DTD.

For the single-node topology, we connect five servers to a switch as shown in Figure 5. All the links have equal bandwidth of 10 Gbps and a delay of $3\mu s$. We later use this simple star topology to form a classic incast traffic pattern to observe the bandwidth share of individual flows and evaluate the fairness of the scheduler.

We built a three-level fat-tree topology for large-scale simulation. As shown in Figure 6, there are 4 Core switches, 8 Aggregation switches, 8 Top-of-Rack (ToR) switches and 256 servers. The links between servers and ToR switches are 10Gbps with 10 ns delay. Other links have a bandwidth of 40Gbps and a $1\mu s$ propagation delay. We apply Gearbox and other scheduler schemes on every ToR/Aggregation/Core switch to evaluate the FCT performance.

Traffic loads We generate empirical traffic workloads based on the datacenter flow size distribution from an operational datacenter that supports web-search service [2]. The traffic follows a heavy tail distribution as Figure 7 shows. The flows arriving in a Poisson process with different arrival
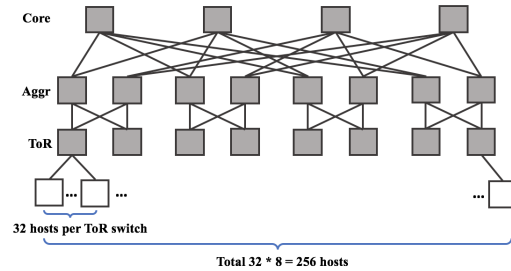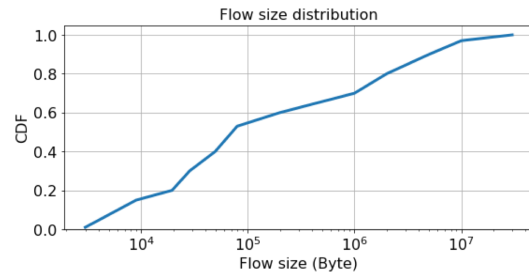
intervals result in different traffic loads. Each flow randomly selects the source and the destination hosts in the topology in a uniform distribution.

Alternative approaches We compare Gearbox with single-level calendar queues [24]. All the schedulers in our evaluation have 56 FIFOs and are evaluated with different granularities as discussed in *section 2.3*. Table 2 provides details of the Gearbox and calendar queue schedulers. We also compared Gearbox with an ideal PIFO-based WFQ scheduler as well as a simple drop tail queue. In our packet-based simulation, all packets have the same size of 1,500 bytes [7] and the number of bytes per virtual time unit is set to 750 bytes (the finest supported granularity).

4.1.2 Single-node Microbenchmark

Gearbox can reach good max-min fairness We observe that Gearbox can reach satisfactory max-min fairness in bandwidth allocation, close to that of PIFO-based WFQ. In

---

[7]In NS2 packet based simulations, packet size is not a factor that affects switch performance.

Table 2: PACKET SCHEDULER SET UP

Packet Scheduler	Granularity
Gearbox	$g_1 = 1, g_2 = 8, g_3 = 64$
CQ-1	$g = 1$
CQ-10	$g = 10$
CQ-100	$g = 100$
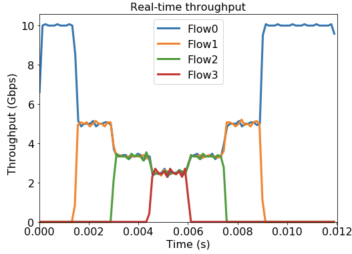
Figure 8: Real-time throughput of Gearbox

Table 3: Flow Weight Sets

Weight Set 1	$1:1:1:1$
Weight Set 2	$2:2:1:1$
Weight Set 3	$50:50:1:1$
Weight Set 4	$100:100:1:1$

our single-node microbenchmark evaluation, we used four TCP flows, each sending traffic to one destination node with different starting times and ending times. Based on the results shown in Figure 8, Gearbox allows the TCP flows to reach a bandwidth max-min fairness with good flow isolation. When a new flow joins, Gearbox quickly adjusts the bandwidth allocated to all the active flows and reaches the max-min fairness. After a flow ends, Gearbox also allows the remaining flows to ramp up quickly.

Normalized Fairness Metric   To better evaluate fairness in bandwidth allocation, we use the Normalized Fairness Metric (NFM). Shreedhar and Varghese first measured fairness in bandwidth allocation using the Fairness Metric (FM) [26]. FM reflects the maximum difference in the number of served bytes between two flows during a time period. By definition, FM's values vary significantly based on the bandwidth assigned to each flow and the total shared bandwidth.

To normalize the influence of this factor, Brent Stephens introduced a better metric: the Normalized Fairness Metric (NFM) [29]. NFM normalizes the Fairness Metric according to the number of bytes that each flow should serve. In a scenario where 4 flows are sharing a link of 4 Mbps with equal weights, each flow is assigned a bandwidth of 1 Mbps. During 1 second, each flow should have 128 kbytes of data served. If the measured Fairness Metric $FM(1sec) = 32kbytes$, then the Normalized Fairness Metric $NFM(1sec) = 32kbytes/128kbytes = 0.25$.

Gearbox has a good NFM even for short time scales. By definition, a lower NFM indicates better fairness in bandwidth allocation. In our evaluation, we observed the NFM of flows with 4 different weight sets (shown in Table 3) in different time scales. From the results shown in Figure 9, Gearbox outperforms coarse-grained calendar queues with different bandwidth allocations. When flows are assigned with weights that differ significantly, the fine-grained calendar queues suffer from packet loss while Gearbox maintains good fairness performance. From the perspective of max-min fairness, Gearbox's performance matches closely that of PIFO-based WFQ.

4.1.3   Large-scale Simulation

We extend our simulation to a three-level fat-tree topology with more servers and higher link rates as described in section 4.1.1. We focus on the normalized FCT performance of different size flows. [8]

Figure 10(a) shows the average normalized FCT across different traffic loads and Figures 10(b) and 10(c) show the average normalized FCT of different size flow groups under 70% and 90% load, respectively. The 95th percentile normalized FCT under various traffic loads is shown in Figure 10(d). Figures 10(e) and 10(f) show the 95th percentile normalized FCT broken down per flow size under 70% and 90% traffic load, respectively.

Short flows benefit from low DTD   Gearbox closely approximates WFQ and provides per-flow isolation, which results in low DTD, a key factor for short flow FCT performance. With WFQ, different flows are isolated from each other and packets from large flows will not block packets from small flows. As we discussed in *section 2.3*, the lower level of Gearbox provides a fine scheduling granularity, guaranteeing that packets from short flows depart according to their departure times without a large discrepancy. Figures 10(b), 10(c), 10(e) and 10(f) show that short flows that are less than 80 kbytes have a small normalized FCT. According to Figure 10, Gearbox can achieve a low normalized FCT close to ideal PIFO-based WFQ and the calendar queue with the finest granularity. On the other hand, packets from short flows would suffer a large DTD in the coarse-grained calendar queues and the drop tail queues.

Short flows consist only of a few packets. Therefore, DTD can cause delays that has a negative effect on their FCT. To further observe the delay of short flows under different scheduler schemes, we measure the average end-to-end delay. As shown in Figure 11, the extra delay in the coarse-grained calendar queues leads to a large normalized FCT for short flows. In contrast, Gearbox has a low delay that is close to that of PIFO-based WFQ.

As stated in *section 3.4*, Gearbox guarantees a low normalized DTD for all packets. We evaluated the normalized DTD of Gearbox [9] as shown in Figure 12. The normalized delay shown in the figure is the actual delay normalized to

---

[8]"Normalized FCT" means a flow's actual FCT normalized to its ideal FCT when no other flows are active in the network.

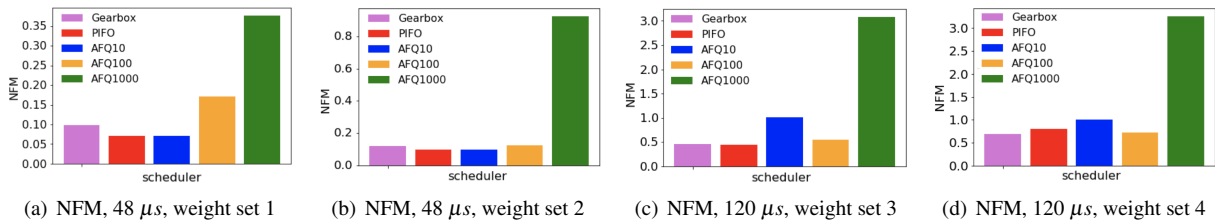[9]Normalized delay was measured in actual (not virtual) time

(a) NFM, 48 μs, weight set 1   (b) NFM, 48 μs, weight set 2   (c) NFM, 120 μs, weight set 3   (d) NFM, 120 μs, weight set 4

Figure 9: Normalized Fairness Metric

(a) Average normalized FCT

(b) Breakdown normalized FCT, 70% load

(c) Breakdown normalized FCT, 90% load

(d) 95th percentile normalized FCT

(e) $95th$ percentile normalized FCT - per flow size, 70% load

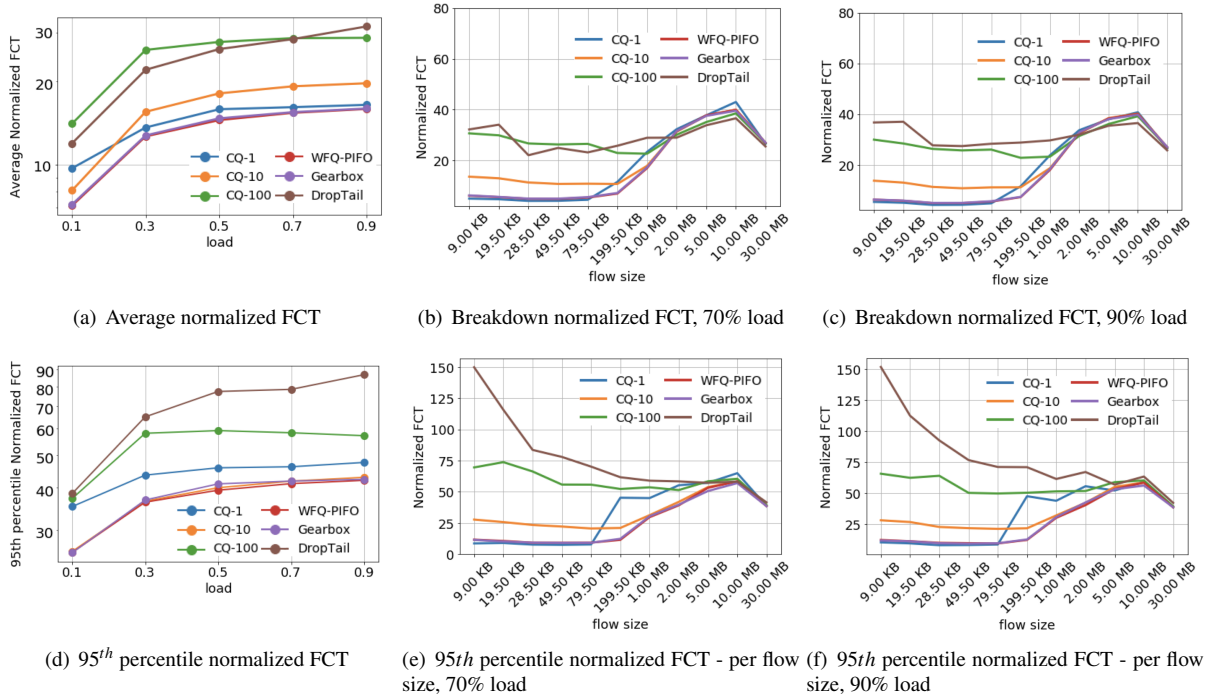(f) $95th$ percentile normalized FCT - per flow size, 90% load

Figure 10: Normalized FCT in fat-tree topology

the delay of the ideal WFQ using a PIFO. In Figure 12, the red dashed line represents the normalized delay with a value of 1, which indicates that the delay is equal to that of ideal WFQ. The simulation results show that Gearbox has a satisfactory normalized delay that is close to 1 for flows with different sizes, indicating that Gearbox has a delay performance closely matching that of ideal WFQ.

Large flows benefit from low packet loss rate Gearbox not only provides a small normalized FCT for short flows, but it also reduces packet loss and re-transmission for large flows. Thanks to its higher levels, Gearbox can schedule packets with large departure times with a very low drop rate.

Figure 10 shows Gearbox can achieve good normalized FCT performance for mid-sized flows around 200 kbytes or larger. We further measured the average packet loss of different flow groups. As shown in Figure 13, the single-level calendar queues with fine granularity drop packets frequently for the mid-sized and large flows, triggering a large number

of re-transmissions and leading to a high normalized FCT. However, Gearbox and other coarse-grained calendar queues have low packet loss rates that are close to zero. Gearbox reduces packet loss related to calendar range overflows and guarantees a low normalized FCT for mid-sized and large flows.

As shown in the above simulation results, Gearbox combines the benefits of both fine and coarse calendar queue granularities. Consequently, Gearbox provides satisfactory normalized FCT performance for flows with different sizes as discussed in *Section 3*. The evaluation results show that Gearbox provides performance comparable to PIFO-based WFQ.

4.2 Hardware Prototype Design

Overview We implemented Gearbox in VHDL with multiple parameters (generics) for easy scalability including:
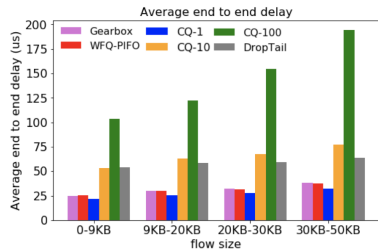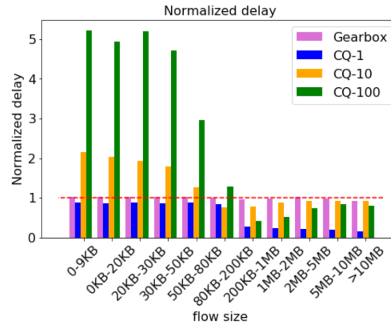
Figure 11: Average end-to-end delay of short flows
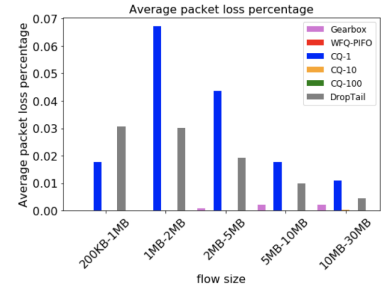

Figure 12: 99th percentile normalized delay

Figure 13: Average packet loss percentage

- Number of levels
- Number of FIFOs per level
- Number of flows
- Other sizing parameters for memories and logic

The VHDL code, test bench, and FPGA implementation files are available at `https://github.com/Gearbox-NSDI/Gearbox_NSDI`

Hardware prototype architecture A high-level block diagram of the VHDL implementation of Gearbox is shown in Figure 14.
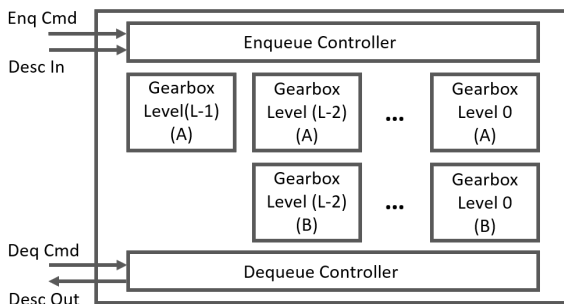
Figure 14: Gearbox Block Diagram - VHDL Implementation

At the top level, Gearbox consists of a parameterized number of instances of the Gearbox Level sub-block along with the Enqueue and Dequeue Controller blocks. Each level except the highest consists of two Gearbox Level sub-blocks denoted (A) and (B). The A and B sub-blocks form a ping-pong scheme to guarantee access to a full set of FIFOs (corresponding to virtual time units) while maintaining packet order. The highest level consists only of a single Gearbox Level sub-block because a wraparound of the FIFO index does not cause out-of-order packets.

The Gearbox Level sub-block shown in Figure 15 consists of a parameterized number of FIFOs along with Enqueue and Dequeue logic blocks.

The enqueue and dequeue operations are described below. The packet descriptor is formatted as follows:

Packet Pointer (15)	Address of packet in packet buffer
Packet Length (11)	Packet length in bytes
Packet Time (20)	Packet transmission time, i.e., the time it takes to transmit the packet at the given flow rate
Flow ID (10)	Flow identification number
Packet ID (16)	Packet identification number (used only to detect out of order events)
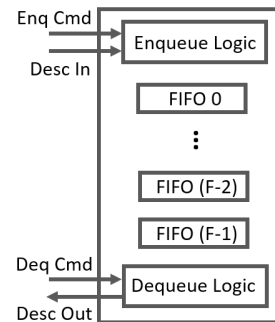
Figure 15: Gearbox Level - VHDL Implementation

The width of each descriptor subfield is parameterized. The numbers in parentheses denote example widths for a given configuration.

Enqueue Operation Upon receipt of an enqueue command, Gearbox performs the following steps:

1. Calculate the packet's departure time based on the packet transmission time and the system time t_s

2. Determine the enqueue level and enqueue FIFO within that level

3. Store the packet descriptor in the calculated level and FIFO

Gearbox completes an enqueue operation in three clock cycles.

Dequeue Operation   Upon receipt of a dequeue command, Gearbox performs the following steps:

1. Find first non-empty level and FIFO and update the system time t_s

2. Calculate number of bytes to serve from each level

3. Dequeue from each level a number of descriptors to reach or exceed the number of bytes to serve

Gearbox completes a dequeue operation in four clocks when performing steps 1 through 3 and in two clock cycles if performing only step 3 (i.e., dequeuing from each non-empty level once steps 1 and 2 are completed).

Math Operations   The VHDL implementation of Gearbox is scalable using generics that are powers of 2, notably for the number of levels, granularity of each level, and the number of FIFOs within each level. This enables calculations that require division operations to be implemented using bit shifting or truncation, which are much more efficiently implemented in logic gates.

Targeting to an FPGA   We targeted the Gearbox VHDL design configured with 4 levels and 16 FIFOs per level, 256 locations (packet descriptors) per FIFO, and 1K flows to a Xilinx Alveo U250 board [34], which uses an UltraScale+ VU13P FPGA with mid-speed grade. Using Vivado 2020.2 [33], we obtained the utilization and performance shown in Table 4.

Table 4: FPGA prototype utilization and performance

	Frequency	LUTs	FFs	BRAM
Units	350 MHz	12331	9953	96
Device Util Pct		0.71%	0.29%	3.6%

With an enqueue and a dequeue every four clocks, the design sustains a packet rate of $350 \div 4 = 87.5$ Mpkts/sec, which is equivalent to a line rate of 100 Gigabit Ethernet for 123-byte packets or larger, taking into account a Preamble of 8 bytes and an IFG of 12 bytes.

5   Related Work

After the proposal of numerous bandwidth allocation algorithms such as WFQ, PGPS, and SCFQ, academia and industry have worked to implement packet schedulers supporting these algorithms. This trend first begins with the ASIC design. In the 1990s, the Sequencer [8] [9] [10] was an ASIC-based hardware packet scheduler that sorts packets into ascending order based on their departure time with limited scalability. In the 2000s, a specialized data structure:

pipeline heap (P-heap) [4] [17] provided fine-grained priority queues in hardware, which improves the scalability but is required for each egress port [27] [28] and therefore uses significant chip area on high-speed switches. The recent work PIFO [27] [28] provides a programmable packet scheduler, which has a very small chip area overhead and is relatively easy to implement. However, it requires special hardware support (such as TCAM) and has limited scalability.

The limitations of the ASIC-based hardware packet schedulers we mentioned in section 2 motivated research in approximate schedulers based on strict-priority queues. Among them are the Approximate Fair Queuing (AFQ) [24] and Programmable Calendar Queues (PCQ) [25]. AFQ and PCQ perform well in bandwidth allocation with the same weights. However, when flow weights vary over a wide range, AFQ and PCQ's fixed granularity leads to low scheduling precision or packet drops. The authors of PCQ discuss a hierarchical architecture in their paper, but its dequeuing scheme might lead to starvation of flows in the lower level. SP-PIFO [1] is another recent work that uses a unique algorithm to adjust the priority between FIFOs to minimize scheduling errors. However, SP-PIFO may cause misordering of packets within a single flow, which would lead to problems for TCP-based flows.

6   Conclusion

In this paper, we propose Gearbox, a hierarchical packet scheduler that practically approximates WFQ. Gearbox is a FIFO-based packet scheduler targeted to next-gen programmable switches and smart NICs. Its tiered granularity allows Gearbox to achieve an adequate normalized DTD and FCT performance with a relatively small number of queues. Gearbox eliminates packet recirculation and has a streamlined operation that allows it to achieve high packet processing speed. From our evaluation, Gearbox achieves weighted max-min fairness in bandwidth allocation and FCT performance comparable to that of ideal WFQ. We implement Gearbox in an NS2 simulator and a VHDL-based hardware prototype, targeting a Xilinx ALVEO U250 FPGA card. Our Gearbox hardware prototype runs at 350 MHz, which is equivalent to a maximum throughput of 58.8 Gbps with 64-byte packets over 100GbE and full line rate 100GbE with packets larger than 123 bytes.

Acknowledgements

References

[1] ALCOZ, A. G., DIETMÜLLER, A., AND VANBEVER, L. Sp-pifo: Approximating push-in first-out behaviors using strict-priority queues. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)* (2020), pp. 59–76.

[2] ALIZADEH, M., YANG, S., SHARIF, M., KATTI, S., MCKEOWN, N., PRABHAKAR, B., AND SHENKER, S. pfabric: Minimal near-optimal datacenter transport. In *ACM SIGCOMM Computer Communication Review* (2013), vol. 43, ACM, pp. 435–446.

[3] ALJAEDI, A., CHOW, C. E., ELGZIL, A., ALAMRI, N., AND BAHKALI, I. Network virtualization with openflow for large-scale datacenter networks. *IJCSNS 17*, 9 (2017), 10.

[4] BHAGWAN, R., AND LIN, B. Fast and scalable priority queue architecture for high-speed network switches. In *Proceedings IEEE INFO-COM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No. 00CH37064)* (2000), vol. 2, IEEE, pp. 538–547.

[5] BROADCOM. Broadcom StrataDNX™ BCM88480 Traffic Management Architecture. https://docs.broadcom.com/doc/88480-DG1-PUB, 2021.

[6] BROADCOM. High Capacity StrataXGS®Trident II Ethernet Switch Series. http://www:broadcom:com/products/Switching/Data-Center/BCM56850-Series., 2021.

[7] BROWN, R. Calendar queues: a fast 0 (1) priority queue implementation for the simulation event set problem. *Communications of the ACM 31*, 10 (1988), 1220–1227.

[8] CHAO, H. J. Architecture design for regulating and scheduling user's traffic in atm networks. In *ACM SIGCOMM Computer Communication Review* (1992), vol. 22, ACM, pp. 77–87.

[9] CHAO, H. J., CHENG, H., JENQ, Y.-R., AND JEONG, D. Design of a generalized priority queue manager for atm switches. *IEEE Journal on Selected Areas in Communications 15*, 5 (1997), 867–880.

[10] CHAO, H. J., JENQ, Y.-R., GUO, X., AND LAM, C.-H. Design of packet-fair queuing schedulers using a ram-based searching engine. *IEEE Journal on Selected Areas in Communications 17*, 6 (1999), 1105–1126.

[11] CISCO. Cisco Silicon One Product Family White Paper. https://www.cisco.com/c/en/us/solutions/silicon-one.html, 2021.

[12] CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. *Introduction to algorithms*. MIT press, 2009.

[13] DALTON, M., SCHULTZ, D., ADRIAENS, J., AREFIN, A., GUPTA, A., FAHS, B., RUBINSTEIN, D., ZERMENO, E. C., RUBOW, E., DOCAUER, J. A., ET AL. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)* (2018), pp. 373–387.

[14] DEMERS, A., KESHAV, S., AND SHENKER, S. Analysis and simulation of a fair queueing algorithm. In *ACM SIGCOMM Computer Communication Review* (1989), vol. 19, ACM, pp. 1–12.

[15] FIRESTONE, D., PUTNAM, A., MUNDKUR, S., CHIOU, D., DABAGH, A., ANDREWARTHA, M., ANGEPAT, H., BHANU, V., CAULFIELD, A., CHUNG, E., ET AL. Azure accelerated networking: Smartnics in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)* (2018), pp. 51–66.

[16] HASAN, J., CHANDRA, S., AND VIJAYKUMAR, T. Efficient use of memory bandwidth to improve network processor throughput. *ACM SIGARCH Computer Architecture News 31*, 2 (2003), 300–313.

[17] IOANNOU, A., AND KATEVENIS, M. G. Pipelined heap (priority queue) management for advanced scheduling in high-speed networks. *IEEE/ACM Transactions on Networking (ToN) 15*, 2 (2007), 450–461.

[18] KIM, D., YU, T., LIU, H. H., ZHU, Y., PADHYE, J., RAINDEL, S., GUO, C., SEKAR, V., AND SESHAN, S. Freeflow: Software-based virtual rdma networking for containerized clouds. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (2019), pp. 113–126.

[19] MEDEIROS, B., SIMPLICIO, M. A., AND ANDRADE, E. R. Designing and assessing multi-tenant isolation strategies for cloud networks. In *2019 22nd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)* (2019), IEEE, pp. 214–221.

[20] NAGARAJ, K., BHARADIA, D., MAO, H., CHINCHALI, S., ALIZADEH, M., AND KATTI, S. Numfabric: Fast and flexible bandwidth allocation in datacenters. In *Proceedings of the 2016 ACM SIGCOMM Conference* (2016), ACM, pp. 188–201.

[21] NETWORK SIMULATOR DEVELOPMENT GROUP, T. The network simulator ns 2. In *https://www.isi.edu/nsnam/ns/.* 2000, 2000, p. 1.

[22] PAREKH, A. K., AND GALLAGER, R. G. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM transactions on networking*, 3 (1993), 344–357.

[23] PAREKH, A. K., AND GALLAGER, R. G. A generalized processor sharing approach to flow control in integrated services networks: the multiple node case. *IEEE/ACM transactions on networking 2*, 2 (1994), 137–150.

[24] SHARMA, N. K., LIU, M., ATREYA, K., AND KRISHNAMURTHY, A. Approximating fair queueing on reconfigurable switches. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)* (2018), pp. 1–16.

[25] SHARMA, N. K., ZHAO, C., LIU, M., KANNAN, P. G., KIM, C., KRISHNAMURTHY, A., AND SIVARAMAN, A. Programmable calendar queues for high-speed packet scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)* (2020), pp. 685–699.

[26] SHREEDHAR, M., AND VARGHESE, G. Efficient fair queuing using deficit round-robin. *IEEE/ACM Transactions on networking*, 3 (1996), 375–385.

[27] SIVARAMAN, A., SUBRAMANIAN, S., AGRAWAL, A., CHOLE, S., CHUANG, S.-T., EDSALL, T., ALIZADEH, M., KATTI, S., MCKEOWN, N., AND BALAKRISHNAN, H. Towards programmable packet scheduling. In *Proceedings of the 14th ACM workshop on hot topics in networks* (2015), ACM, p. 23.

[28] SIVARAMAN, A., SUBRAMANIAN, S., ALIZADEH, M., CHOLE, S., CHUANG, S.-T., AGRAWAL, A., BALAKRISHNAN, H., EDSALL, T., KATTI, S., AND MCKEOWN, N. Programmable packet scheduling at line rate. In *Proceedings of the 2016 ACM SIGCOMM Conference* (2016), ACM, pp. 44–57.

[29] STEPHENS, B., SINGHVI, A., AKELLA, A., AND SWIFT, M. Titan: Fair packet scheduling for commodity multiqueue nics. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (2017), pp. 431–444.

[30] THIMMARAJU, K., RÉTVÁRI, G., AND SCHMID, S. Virtual network isolation: Are we there yet? In *Proceedings of the 2018 Workshop on Security in Softwarized Networks: Prospects and Challenges* (2018), pp. 1–7.

[31] VARGHESE, G., AND LAUCK, A. Hashed and hierarchical timing wheels: efficient data structures for implementing a timer facility. *IEEE/ACM transactions on networking 5*, 6 (1997), 824–834.

[32] WANG, Z., HUANG, H., ZHANG, J., AND ALONSO, G. Shuhai: Benchmarking high bandwidth memory on fpgas. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (2020), IEEE, pp. 111–119.

[33] XILINX. Vivado Design Suite, Integrated Design Environment. `https://www.xilinx.com/products/design-tools/vivado.html`, 2021.

[34] XILINX. Xilinx Alveo U250 Data Center Accelerator Card. `https://www.xilinx.com/products/boards-and-kits/alveo/u250.html`, 2021.

Appendix

A Step-down FIFO

The solution to the packet out-of-order issue in Section 3.6 may increase the DTD of specific flows. Normally, packets will enqueue at the appropriate level with a granularity that is appropriate for their expected delays. As long as the packets from a flow arrive at the allocated rate, its packets will always enqueue at the appropriate level with a guaranteed low DTD. However, when a flow's input data rate exceeds its allocated bandwidth, its packets may queue up at higher levels. As the scheme in Section 3.6 maintains the 'Last packet enqueued level' L_i, such flows would only enqueue their subsequent packets in the higher levels from that point on. As previously discussed, when packets that belong to a lower level enqueue at a higher level, they suffer from a coarser granularity and higher DTD. This leads to larger packet delays and increases FCT. In this case, flows need to step back to the lower level in which they are supposed to enqueue so they can restore the lower DTD. Is it possible for a flow that queues up to a higher level to step down to a lower level when its arrival rate decreases to its admitted bandwidth?

The answer is yes: we introduce the 'Step-down FIFO', which allows flows at a higher level to go to a lower level. To understand the design of the Step-down FIFO, we must first understand why we need to enqueue packets into a higher level. According to Section 3.4, we can secure the serving order of packets as long as we serve them with the same granularity. In other words, if we can serve packets at the higher level with the same granularity at the lower level, it would be safe to enqueue the subsequent packets into the lower level without causing packet misordering. At this point, 'Step-down FIFO' serves as a special FIFO at a higher level that provides the same granularity of a lower level. A 'Step-down FIFO' expands a FIFO at the higher level into multiple queues with finer granularity, preserving the departure
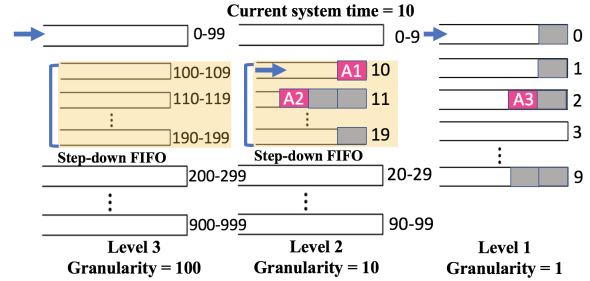

Figure 16: Step-down FIFO

time difference between the packets. When a flow's latest packet enqueues into a 'Step-down FIFO' at level l, Gearbox schedules it with the granularity of level $(l-1)$. Thus, we can mark the 'Last packet enqueued level' $L_i = (l-1)$. As a result, we can enqueue the subsequent packets from the same flow into level $(l-1)$.

Figure 16 illustrates how a 'Step-down FIFO' works. FIFOs marked in yellow at level 2 and 3 are 'Step-down FIFOs', which maintain the same granularity as the next lower level. In this example, Gearbox uses 10 FIFOs to achieve a Step-down FIFO with the finer granularity. Packets $A1$ and $A2$ enqueue the 'Step-down FIFO' and preserve their departure times $F_{(A,1)} = 10$ and $F_{(A,2)} = 11$ with the granularity of $g_1 = 1$. By doing so, Gearbox will schedule packets $A1$ and $A2$ with the same granularity at level 1 and the packets will leave the scheduler at $t_s = 10$ and 11. In this case, it is safe to place packet $A3$ at level 1 without causing packet out-of-order issues. In this example, 'Step-down FIFO' makes it possible for flow A to step down from level 2 to level 1.

With a 'Step-down-FIFO', the flows that follow their allocated rate will eventually get back to the level they belong to and restore their DTD. Assume flow i belongs to level l, where the departure time of its packets increases by g_l. Due to prior burstiness, packets from flow i queue up at a higher level l' and need to step back down to level l. Since flow i now follows its admitted rate r_i, its packets arrive with a departure time interval of g_l without accumulation. Based on the basic idea of Gearbox in Section 3.1, each FIFO at level l' (including the 'Step-down FIFO') covers a departure time range of $g_{l'}$ and $g_{l'} >> g_l$. Since $F_{(i,k)}$ increases by g_l, eventually there will always be a packet that enqueues in the 'Step-down FIFO' at level l'. As the 'Step-down FIFO' at level l' schedules packets with the granularity of g_l, Gearbox will mark the last enqueue level of flow i as $L_i = l$ and the subsequent packets from flow i will get back to level l. In this way, the Step-down-FIFO enables the flows that follow their allocated rates to eventually get back to the lower level to which they belong.
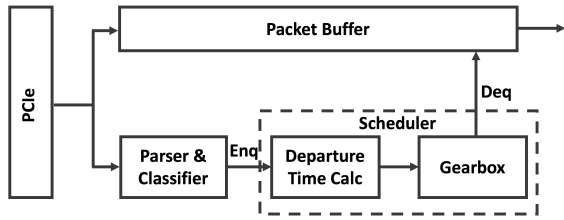
Figure 17: Gearbox's Application in a NIC (egress only shown)
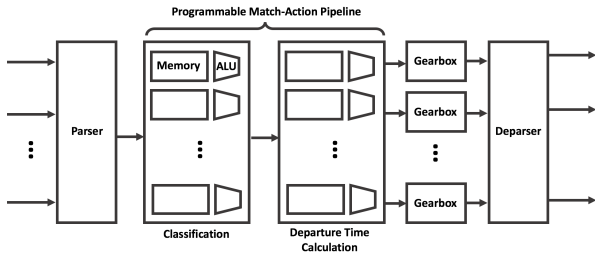
Figure 18: Gearbox's Application in Multi-pipeline Systems

B   Gearbox's Application in NIC and Multi-pipeline Systems

Figure 17 shows a Gearbox application in a NIC with only the egress path shown. Packets from the server arrive over PCIe and are stored in the Packet Buffer. The packet headers are parsed and classified in the Parser & Classifier block to extract the flow id and the packet length. The Departure Time Calculator computes the departure time using the packet length and forwards it to Gearbox for enqueue. Gearbox dequeues descriptors and forwards the packet pointers to the Packet Buffer to output the packets.

Figure 18 shows a Gearbox application in a multi-pipeline switch. After the parsing stage, the classification (flow identification) is done in the first stage. The second stage computes the departure times, which are fed to multiple Gearboxes for enqueue. Dequeued packets are output by the deparser.

Performance Interfaces for Network Functions

Rishabh Iyer, Katerina Argyraki, George Candea

EPFL, Switzerland

Abstract

Modern programmers routinely use third-party code, and infrastructure operators deploy software they did not write. This would not be possible without semantic interfaces—documentation, header files, specifications—that succinctly describe what that third-party code does.

We propose *performance interfaces* as a way to describe a system's performance, akin to how a semantic interface describes its functionality. We concretize this idea in the domain of network functions (NFs) and present a tool (PIX) that automatically extracts performance interfaces from NF implementations. We evaluate PIX on 12 NFs, including several used in production. The resulting performance interfaces are accurate yet orders of magnitude simpler than the code itself and take minutes to extract. We show how developers and operators can use performance interfaces to identify performance regressions, diagnose and fix performance bugs and identify the latency impact of NIC offloads.

PIX is available at https://github.com/dslab-epfl/pix.

1 Introduction

Semantic interfaces (e.g., abstract classes, specifications, header files, documentation) succinctly describe a program's externally visible functional behavior, enabling engineers to use the system productively. This makes it possible for programmers to use a lot of third-party code and makes infrastructure operators comfortable with deploying software they did not write.

We do not know of an equivalent construct for describing performance behavior in a way that is simultaneously succinct, precise, complete, and human-readable. Engineers reason about performance in terms of envelopes (e.g., "runs in $O(n)$ time") and benchmarks, which implies that they deploy their system without understanding the entire spectrum of performance it can exhibit. As a result, untested inputs can exercise mysterious code paths that lead to unexpected performance behavior [4, 36, 39] and a perpetual need to fix performance bugs [35, 43].

In this paper, we explore the idea of a *performance interface*: a description of a system's performance behavior that is simultaneously succinct, precise, and human-readable. What should such an interface look like? Like a good semantic interface, it should be "much smaller and simpler than the code" [48], so it must abstract away certain details—but which ones? Performance problems often lie in low-level implementation details as well as the code's interaction with the environment (e.g., specifics of the underlying hardware's cache hierarchy). Is it possible to capture all the relevant performance behaviors of a system while being "much smaller and simpler" than the system itself?

We propose that the performance interface of a system be a *program* that accepts the same inputs as the system and outputs how long the system would take to process the given input[1]. A performance interface has a *resolution*, which quantifies the smallest change in performance that it specifies (e.g., 50 ns, 1 mem-op)—the coarser the resolution, the simpler the interface. We distinguish a *deployment-specific* interface from a *general-case* one: The former is much simpler and of greater interest to an operator, who wants to understand the system's performance behavior in her specific environment, while the latter is most useful to developers. This distinction, along with resolution, makes it possible to have performance interfaces that capture only those behaviors that are relevant to the case at hand. In other words, the two concepts enable abstraction of performance behavior.

We concretize our proposal in the context of network functions (NFs), i.e., load balancers, firewalls, NATs, etc. NFs are typically on the critical path of serving a user request and often face unpredictable traffic coming from the outside world. For instance, any packet that enters a service provider's data center traverses at least one load balancer/reverse proxy and typically also a firewall—the latency that each NF adds to the packet directly impacts the user-perceived latency. A recent survey [54] of network operators found NF performance degradation to be a frequent pain point, and such performance bugs to be among the hardest to diagnose.

To make NF performance interfaces useful today, we developed PIX (**P**erformance **I**nterface e**X**tractor). PIX takes as input NF code written in C and outputs general-case performance interfaces in the form of small Python programs that it can then specialize into deployment-specific interfaces for individual deployments. PIX currently supports three latency-related metrics: number of instructions, number of memory operations, and number of CPU cycles. For each metric, PIX outputs one set of Python programs; each set contains one Python program per relevant range of resolutions. All PIX-extracted performance interfaces are specific to the CPU's ISA.

---

[1] In this paper we focus on system latency, not throughput.

Further, the interfaces for CPU cycles are specific to the micro-architecture of the underlying hardware and assume that the NF does not contend for hardware resources with other processes (i.e., assume either smart process co-location [12, 31, 52] or process isolation, using techniques such as cache partitioning [77]). Under the covers, PIX employs symbolic program analysis techniques to reason about the NF's performance behaviors.

We evaluate PIX on 12 open-source NFs, including the Katran load balancer [71] used at Facebook, the Natasha NAT [58] used at Scaleway and the XDP packet filter from the Cilium project [14]. All 12 NFs were written using either the Linux kernel's eBPF XDP [82] framework or the DPDK [21] kernel-bypass framework, two of the most popular ways to develop high-performance NFs. Our evaluation shows that the extracted performance interfaces are accurate yet orders of magnitude simpler than the code, and take minutes to obtain. We show how performance interfaces extracted by PIX can be used to identify performance regressions, diagnose and fix performance bugs, and identify the latency impact of NIC offloads.

In summary, we make two contributions in this paper:

- We propose the concept of performance interfaces, which leverages the notions of performance resolution and deployment-specific interfaces to enable abstraction of performance behavior.
- We demonstrate that it is feasible to build a tool that automatically extracts performance interfaces from NF code, and that these interfaces can be accurate-yet-simple enough to help understand and debug performance.

In the rest of the paper, we describe how we think a performance interface should look like (§2). Then, we describe PIX (§3) and use it to evaluate the feasibility and utility of performance interfaces for NFs (§4). Finally, we discuss how PIX can generalize to systems beyond NFs (§5), related work (§6), and conclude (§7).

2 Performance Interfaces

In this section, we present our proposal for performance interfaces, and describe how we envision them being used.

Target audience: We target two categories of audience for any system: The *developers* write the code for the system and are familiar with its low-level implementation details, but not necessarily with all possible performance behaviors it can exhibit. The *operators* did not write the code but instead seek to use/deploy/build on top of the system in their respective environments. They

are unfamiliar with and do not necessarily want to understand its low-level details. Further, unlike the developers who care about the system's performance in all settings, they care primarily about its performance in their specific use-case/deployment. These categories can vary from system to system—the developer of an application A might themselves be building upon on a network stack B, making them an operator for that stack.

Design goals: We envision that a "performance interface" must describe the system's externally visible *performance* behaviors, just as a semantic interface describes a system's externally visible functionality [48].

The primary challenge in summarizing performance is that systems typically expose a greater variety of performance behaviors than semantic ones. Hence, a performance interface that perfectly predicts every possible performance behavior would likely be so complex that it wouldn't deserve to be called an interface.

We look for a compromise, i.e., a way to summarize performance that achieves a good balance between the following properties: (1) **Accuracy**, i.e., the ability to summarize performance completely (for every possible input) and precisely (with a small error). (2) Simplicity, i.e., being **smaller** than the code and as **abstract** as possible—summarize performance in terms of primitives appropriate for a semantic interface of the system, and reveal implementation details only when necessary.

We also aim for (3) **Portability**. A system's performance may depend significantly on its environment (e.g., workload, hardware). For instance, adversarial traffic causing L3 cache misses can degrade NF latency by $3\times$ [64]. The interface should make it easy to quantify the impact of a particular environment on performance, enabling porting of the interface across deployments.

State of the art: Today, performance is typically summarized through upper bounds—Big-Oh notation or worst-case execution time [79]—and statistics (e.g., x-th percentile latency). These descriptions maximize simplicity at the cost of accuracy—there are many inputs for which they do not provide accurate predictions.

We draw inspiration from two recent proposals that describe a system's performance behavior as performance annotations [69] and performance contracts [41] respectively. Freud [69] describes a method's performance as a performance annotation: a set of ⟨input/global-variable constraints, performance formula⟩ tuples, and each formula is a mathematical function of the method's input and/or global variables. Bolt [41] describes an NF's latency as a "performance contract:" a set of ⟨input constraints, performance formula⟩ tuples, where each formula is a function of the system input and "Performance-Critical Variables" (PCVs).

Since we reuse the idea of PCVs from performance contracts [41], we elaborate upon them here. A PCV is a parameter that captures the influence on performance of all factors other than the input packet (e.g., NF configuration, state built up by prior packets, hardware characteristics, etc). A PCV is not always an explicit variable in the NF implementation, rather it can be an implicit "ghost" variable [29, 32]. For instance, if an NF employs a hash table, a PCV could be the "number of collisions" encountered by the current packet—this ghost variable allows latency to be expressed as a function of, among other things, the number of collisions. Independent prior work [37, 38] on symbolic bounds has also argued for the use of a PCV-like abstraction to succinctly summarize the performance behavior of stateful programs.

However, neither annotations nor contracts were designed to be "performance interfaces". Contracts sacrifice simplicity for accuracy—they include a list of input constraints, which can be as many as the number of execution paths through the system and reveal low-level implementation details *even when unnecessary*. Annotations do not meet our accuracy goal as they do not capture how performance depends on state built from past inputs, e.g., the contents of a hash table or a hardware cache (Appendix A).

Definition: The *performance interface* of a program P with procedures $p_1, p_2, ...$ is a program $S_P = \{p'_1, p'_2, ...\}$. A procedure $p'_i \in S_P$ takes the same inputs as the corresponding $p_i \in P$ and returns the performance of executing p_i. This return value corresponds to a performance metric (e.g., # of `x86` instructions, # of CPU cycles). The *resolution r* of S_P is the smallest difference in performance that S_P can specify: if $\mathcal{P}(p_i(I))$ is p_i's performance given input I, then $|p'_i(I) - \mathcal{P}(p_i(I))| < r, \quad \forall p_i, I$.

A performance interface can be for the "general case" or specific to a deployment.

In a *general-case performance interface*, the procedures p'_i compute performance as a function of PCVs [41]. PCVs ensure that the interface can describe the performance of each p'_i in full generality, i.e., for arbitrary workloads and hardware configurations.

A *deployment-specific performance interface* is simpler than the general-case one and does not contain PCVs. Instead, procedure p'_i returns performance as a statistic (e.g., median, max, 99th percentile), computed for a given joint probability distribution of the PCVs that describes P's environment for a particular deployment. In this work, an NF's deployment environment is defined by its configuration read at startup, a representative workload, and the specific hardware it runs on.

Example: We illustrate with an example implementation of a MAC learning bridge (Fig. 1) that uses a

```
void bridge(pkt* p, time_t now) {
  expire_stale_ports(now);
  if (invalid_hdr(p)) {
    DROP(p);
    return;
  }
  /* Learning source MAC addr */
  if(!slow_MACtable_get(p->src_mac, &p->port))
    slow_MACtable_put(p->src_mac,&p->port);
  else
    slow_MACtable_update(p->src_mac, now);
  /* Forwarding based on dest MAC addr */
  if (fast_MACtable_get(p->dst_mac,&out_port))
    FORWARD(p,out_port);
  else if (slow_MACtable_get(p->dst_mac,&
    out_port))
    FORWARD(p,out_port);
  else
    BROADCAST(p,p->port);
}
```

Figure 1. Example implementation of a MAC learning bridge

fast MAC table, implemented in hardware, and a slow software-based table, based on a cuckoo hash table. Table 1 shows the performance cost of this implementation's procedures in terms of executed lines of pseudocode (LOP), a performance metric we use for illustration only. For now, we assume these costs, we elaborate on how they are obtained in §3.

Operation	Performance [LOP]
`expire_stale_ports()`	$40 + 60 \times$ n_stale
`invalid_hdr()`	5
`DROP`	1
`FORWARD`	60
`BROADCAST`	200
`fast_MACtable_get()`	10
`slow_MACtable_get()`	50
`slow_MACtable_update()`	70
`expire_stale_ports()`	$40 + 60 \times$ n_stale
`slow_MACtable_put()`	$110 + 80 \times$ n_evicted $+ 120 \times$ occ $\times$ rehashing

Table 1. General-case performance of procedures called by the code in Fig. 1. Two have non-constant performance: expiring learned ports is linear in the number of stale ports, and doing a `put()` in the cuckoo hash table depends on the number of keys that must be evicted and whether rehashing is necessary.

Fig. 2 shows two performance interfaces of this implementation. Since it exposes a single procedure, the performance interface also has a single procedure. The resolution of the performance interfaces is $r = 50$ LOP.

The general-case interface gives performance as a function of 4 PCVs: number of stale flows (`n_stale`), hash-table occupancy (`occ`), number of hash-table evictions triggered by this input (`n_evictions`), and whether rehashing is needed (`rehashing=1` if yes, 0 otherwise). Since the performance metric LOP is independent of the underlying hardware, all 4 PCVs are specific to the

```python
def perf_bridge_gc(p,now):
  # Metric: LOP, Resolution: 50
  # NF state: slow_MACtable, fast_MACtable

  if invalid_hdr(p):
    return 46 + 60* n_stale
  if fast_MACtable_get(p->dst_mac) or
    slow_MACtable_get(p->dst_mac):
    return 280 + 60* n_stale + 80*
    n_evictions + (120* occ) * rehashing
  else
    return 445 + 60* n_stale + 80*
    n_evictions + (120* occ) * rehashing
```

```python
def perf_bridge_ds(p,now):
  # Metric: LOP, Resolution: 50
  # Statistic: 50th percentile
  # NF state: slow_MACtable, fast_MACtable

  if invalid_hdr(p):
    return 106 #(46+60)
  if fast_MACtable_get(p->dst_mac) or
    slow_MACtable_get(p->dst_mac):
    return 340 #(280+60)
  else
    return 505 #(445+60)
```

Figure 2. General-case (left) and deployment-specific (right) performance interfaces for the bridge (Fig. 1). Each return value in the latter is the median LOP executed for the assumed PCV distribution.

bridge's implementation. If the bridge stored the MAC table using a binary tree instead of a cuckoo hash table, the interface would describe performance using different PCVs (`tree_depth` instead of `rehashing`).

The deployment-specific interface gives the median latency for a deployment where the expected workload is such that 50% of input packets encounter no hash collisions and expire ≤ 1 stale ports. The interface produces concrete numbers corresponding to this deployment-specific PCV distribution. Note, the deployment-specific interface does not restrict the inputs (e.g., the types of packets), it only instantiates the PCVs.

This performance interface captures all the performance behaviors of the bridge that are externally visible at resolution $r=50$. It is accurate, in that it correctly predicts performance (at the given resolution) for every possible input. It is smaller and simpler than the implementation: each procedure considers only three operations (invalid header check, fast table lookup, and slow table lookup), since these are the only ones that affect performance at $r=50$. Unlike the general-case interface, the deployment-specific interface makes assumptions about the expected workload.

Why represent the interface as a Python program?
We believe that an interface that presents performance like the system itself—through code that branches on the input—is more intuitive than a list of input constraints for developers and operators. We chose Python due to its ubiquitous use [33].

Resolution: Often, developers and operators do not care about certain performance differences, either because they do not affect their performance targets, or because they are masked by the environment. For example, developers building minute-scale applications may not care about μs-scale variability in the networking stack, while those building μs-scale ones typically do.

The notion of resolution enables the developer/operator reading the interface to choose between multiple levels of abstraction (trading off accuracy for simplicity) in a controlled manner. A performance interface at a specified resolution only differentiates between input classes whose performance differs by more than the resolution—implementation details that cause variability relevant to the specific developer/operator are abstracted away. In our bridge example, a performance interface with a resolution of 1 LOP must report the performance of each forwarding behavior separately; an interface with resolution $>= 45$ LOP can abstract away the difference between a fast and slow lookup, and an interface with resolution $>= 115$ LOP can abstract away the difference between a successful and unsuccessful lookup.

Picking the right resolution: We envision developers/operators picking their respective resolutions based on the performance variability they are willing to tolerate in their deployment scenarios. In §3, we show how PIX goes a step further for those unsure of the "right" resolution, by identifying a minimal set of resolution thresholds that yield all the possible different performance interfaces. This is possible since the performance interface can only elide each implementation detail at a distinct resolution threshold, which results in it not changing between two such thresholds. In our bridge example, {1, 20, 45, 115, 210} is such a minimal set of resolution thresholds, i.e., other resolutions don't yield different interfaces (e.g., the interface at $r = 50$ is identical to that at $r = 46$). By identifying these resolution thresholds, PIX enables developers and operators to easily pick the resolution (and corresponding interface) that achieves the desired trade-off between accuracy and simplicity.

Deployment-specific interfaces: We chose to have separate general-case and deployment-specific interfaces to

provide a different balance between accuracy and simplicity for operators and developers respectively.

General-case interfaces are meant for developers. Developers cannot always predict where/how their code will be deployed, and are hence often interested in the performance of their system when deployed in arbitrary environments. The general-case interface provides them with such a description by summarizing the impact of the environment on the system's performance using PCVs. While PCVs do reveal implementation details (e.g., n_evicted, rehashing reveal the use of a cuckoo-hash table), these details *are necessary* to summarize performance *for an arbitrary workload*, so they must be represented in the general-case interface.

We designed the deployment-specific interface for operators. Since operators are unfamiliar with the system's implementation and only care about the system's performance behavior in their particular deployment environment, the deployment-specific interface does away with the hard-to-understand PCVs by instantiating them with a distribution specific to that deployment. This enables the deployment-specific interface to summarize performance in an NF-generic way—any NF would normally involve a header check and state lookups—and be understood by almost any NF operator. Of course, it does reveal one important aspect of the implementation, namely the distinct fast and slow tables. However, this aspect (which would have no place in a semantic interface) is crucial to any bridge operator interested in performance.

That said, we do not envision the separation between the general-case and the deployment-specific interfaces being set in stone—developers may refer to the deployment-specific interface to understand performance in the face of specific workloads, while operators may refer to the general-case interface to understand performance beyond their expected workload.

3 Extracting Performance Interfaces

We now describe PIX, which takes as input an NF implemented in C and automatically extracts performance interfaces in the form of Python programs.

We designed PIX to meet two goals: (1) *miniminal developer effort*: developers/operators should not need to write performance test suites or proof lemmas, and (2) *allow for proprietary NFs*: NF vendors typically provide operators with only binaries [55]; it's ok for them to provide a performance interface, but not source code.

Fig. 3 presents an overview of PIX. The NF developer gives the PIX back-end the NF source, augmented with a few single-line annotations akin to instantiating a type in a higher-level language. PIX combines this with a preanalysis of the data structures used by the NF and extracts the general-case interfaces for all meaningful resolution
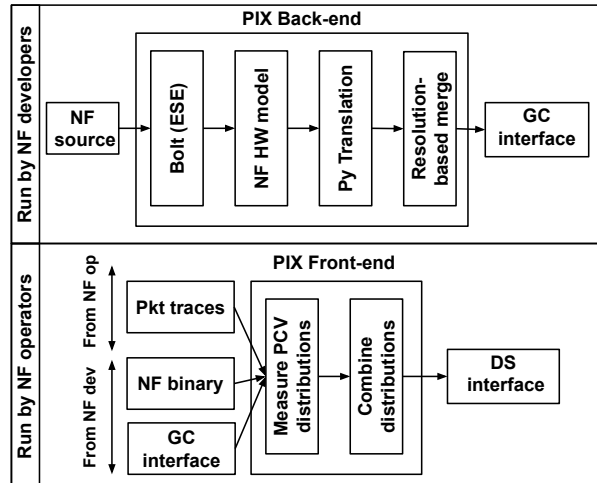

Figure 3. Overview of PIX. GC and DS refer to general-case and deployment-specific respectively.

ranges. The NF operator provides the PIX front-end with an NF binary and general-case interface (provided by the NF developer), along with a (set of) packet trace(s) that represent the expected workload in their deployment. From these, PIX extracts the deployment-specific interfaces for all meaningful resolution ranges. NF developers/operators can also query PIX with a specific resolution, to get the interface at that resolution.

Limitations and Assumptions: The PIX back-end uses exhaustive symbolic execution (ESE) [45] to automatically analyze the NF code. For this to work, the NF needs to be single-threaded, all its loops except the top-level event loop must have statically computable bounds, and it must keep all history-dependent state in data structures with clear interfaces. PIX cannot extract performance interfaces for NFs that do not meet these requirements.

Many (not all) data-plane NFs meet these requirements. For instance, many NFs are written using the eBPF [82] framework as stateless modules that keep their state in cleanly separated, kernel-maintained eBPF maps [24]. Other examples include recently proposed NF frameworks that build upon the DPDK kernel-bypass framework [21] like FastClick [5] and Vigor [83], which impose the use of a specific set of well-separated data structures to store NF state. Counterexamples include Intrusion Detection Systems (IDSes) and TCP-terminating NFs; in general PIX cannot extract interfaces for them and we describe this limitation in more detail in §5.

PIX-extracted interfaces only summarize the processing latency for each packet and do not reason about queuing latencies. Reasoning about these latencies would require PIX to reason about multiple inputs together, and for this, we need to employ techniques more sophisticated than ESE [83]. Reasoning only about processing

latency allows PIX to avoid reasoning about load-based variability since processing latency (unlike queueing latency) does not vary with load.

To capture how hardware affects performance with reasonable accuracy, PIX assumes that the NF runs pinned to a core and does not significantly contend for hardware resources, e.g., due to smart process isolation [12, 31, 52, 77]. We believe network operators keen on predictable performance are likely to employ such techniques.

Implementation: PIX builds on the KLEE symbolic execution engine [9]. We extended KLEE with 4629 lines of C++ code to implement the first two steps of the back-end. We synthesize the Python-based interfaces using 1825 lines of OCaml. We implemented resolution-based merging and the PIX front-end in 1221 lines of Python.

3.1 Extracting general-case interfaces

We now describe how the PIX back-end extracts general-case interfaces from NF source code.

Step 0: Pre-processing: This step outputs the performance of each execution path of the NF in terms of hardware-independent metrics as a function of PCVs specific to the NF's implementation (we refer to these PCVs as *hardware-independent PCVs* henceforth). PIX currently supports two hardware-independent metrics—instruction count and memory-access count. We call this Step 0 because it is not part of our contribution, we largely reuse the approach and tool from Bolt [41].

Bolt relies on the observation that data-plane NFs tend to use the same, relatively few data structures, mainly hash tables/maps and buffers/rings. One can therefore collect these data structures in a library, have an expert "pre-analyze" them once, and then amortize this analysis cost across all NFs that use the library. Bolt's pre-analysis consists of two manual tasks for each call in the library's interface: (1) identify the PCVs relevant to that call, and (2) write a simple symbolic model of the call. Such manual effort is reasonable because it is a rare effort (e.g., once per update to the Linux kernel's eBPF maps) and it is done by the maintainer of the data structure library instead of its users. To illustrate, there were 34 new commits in Linux's eBPF maps last year [23] while the Cilium project [14] alone—just one among hundreds of projects that leverage eBPF maps—had an order of magnitude more commits during that same period [13]. Further, independent prior work [38] has observed that most data structures require only a "few" PCVs, and identifying them is "straightforward". Our experience as the "experts" for this work corroborated this observation—identifying PCVs required only single-line loop annotations which took ≤ 1 person-hour for someone familiar with the data structure code.

The Bolt tool takes as input the NF source code, as well as the symbolic models and loop annotations for the state-accessing calls made by the NF; and outputs the performance of each execution path through the NF as a function of the hardware-independent PCVs.

In Step 0, PIX uses the Bolt tool as stated above, and also automatically instruments the NF code such that it can log the values of the hardware-independent PCVs for each input packet encountered.

Step 1: NF-domain hardware model: This step characterizes the performance of each execution path of the NF in terms of hardware-dependent metrics (CPU cycles), by introducing hardware-dependent PCVs; i.e., PCVs that capture the interaction between NF and hardware.

PIX uses the notion of a CPI (Cycles Per Instruction) stack [27] to compute the number of CPU cycles of an execution path. A CPI stack breaks down the average CPI for a program executing on a given microprocessor into a base CPI plus various CPI components that reflect "lost" cycle opportunities due to miss events such as branch mispredictions and cache/TLB misses. In general, replicating a perfect CPI stack is infeasible—it is equivalent to analyzing each execution path to the depth provided by a cycle-accurate simulator.

We leverage NF-domain knowledge to eliminate CPI components and pick only the necessary set of hardware-dependent PCVs. When an NF runs pinned to a core and with limited contention for hardware resources, the dominant hardware factor that influences its performance is the last-level cache (LLC) [20, 52, 77]. Hence, PIX introduces only two hardware-dependent PCVs—*base_CPI* and *LLC_miss_latency*—and expresses a path's CPU cycle count as *instructions · base_CPI + LLC_misses · LLC_miss_latency*. Note, while PIX uses the same two PCVs for all NFs, the values of these PCVs vary with each *<NF, HW>* pair (§3.2). To track possible LLC misses, PIX leverages taint-analysis [70] to identify independent heap accesses specific to the current input; it then branches on each such access, with one outcome being an LLC miss and the other an LLC hit.

Step 2: Python program: The previous steps specify an NF execution path as a set of symbolic constraints on the input packet and symbols arising from calls to data structures; this step translates these constraints into human-readable python code and outputs a general-case performance interface of the NF with a resolution of 1.

PIX translates symbolic constraints on the input packet using knowledge of the header format of the popular networking protocols (e.g., IPv4, TCP, QUIC). For instance, the constraint $pkt[23 : 24] == 6$ on a non-tunnelled IPv4 packet is translated to $pkt.isTCP$.

```
1 # Developer annotation:
2 DS_INIT(&map,"macTable","ethaddr", struct
      eth_addr,"port", int);
3
4 # Starting condition derived from implem:
5 if bpf_map.unnamed_symbol
6 # Transform based on called library function
7 if bpf_map.contains(pkt[7:12])
8 # Transform based on developer annotation
9 if macTable.contains(pkt.src_mac)
```

Figure 4. Example of PIX's constraint rewriting.

PIX translates symbols arising from calls to data structures using call context and developer-provided annotations (one annotation per instantiated data structure). Fig. 4 illustrates such a translation: Line 2 shows a developer's annotation for a data structure of type `map`: it indicates that this NF uses this map as a `"macTable"`, which maps `"ethaddr"` keys to `"port"` values; these are human-friendly terms chosen by the developer to help the generation of simple performance interfaces. Line 5 shows a constraint derived from the NF code that concerns this map. Line 7 shows how PIX rewrites this constraint because it knows that this is a call to `bpf_map_-lookup_elem()` with an argument corresponding to bytes $7 - 11$ of the input packet. Line 9 shows how PIX further rewrites the constraint because the developer's annotation enables PIX to identify the given bytes as the input packet's source MAC address.

The annotation on Line 2 is the only annotation that the NF developer needs to provide. We believe such one-line annotations are reasonable since they are similar to instantiating a type in a higher-level language.

Step 3: Resolution-based merging: This step uses the notion of resolution to simplify the performance interface: First, it calculates the maximum performance impact of each constraint, i.e., the maximum performance difference between two execution paths that only differ w.r.t this constraint. The set of distinct "maximum performance impacts" forms the minimal set of resolution thresholds. Second, it eliminates all constraints with an impact smaller than the target resolution.

3.2 Extracting deployment-specific interfaces

To extract a deployment-specific interface, the PIX frontend takes as input the NF binary and its general-case interface[2], provided by the NF developer/vendor; along with a (set of) deployment-specific packet trace(s), provided by the NF operator. It then runs the NF binary using the packet trace(s) as input, infers the deployment's PCV

distributions, and instantiates the deployment-specific interface. Running the NF allows PIX to extract accurate deployment-specific interfaces since it can precisely measure the performance impact of the NF's environment as opposed to modeling it.

PIX infers three PCV distributions per NF, deployment:

Hardware-independent PCVs: PIX leverages the instrumentation introduced in Step 0 to measure the values of these PCVs encountered by each packet in the provided trace(s). It then computes a joint probability distribution of these PCVs, since they tend to be highly correlated (e.g., in Table 1, `n_stale` and `n_evictions` are both functions of `occ`).

Base CPI: PIX measures this using hardware performance counters [75] available on all major processors today. Since the packet trace(s) may not exercise all execution paths, PIX assumes the same base-CPI distribution across all paths, and it provides warnings if it detects significant differences (e.g, some paths use expensive x86 instructions, like integer divide, while others don't). We think this is a reasonable assumption because the base CPI is only a function of the instruction mix (it does not include any miss events). In §4.1, we experimentally validate this.

LLC miss latency: Measuring the distribution of LLC miss latency ideally requires sophisticated NF-specific testing [64], to account for the NF's particular instruction- and memory-level parallelism. PIX avoids this because it targets NFs that keep all their state in a relatively small set of pre-analyzed data structures. For each data structure, we craft a microbenchmark that triggers LLC misses.[3] PIX estimates the LLC-miss-latency distribution of each data-structure call in a given deployment, by running the corresponding microbenchmark on the deployment's hardware. In §4.1, we experimentally show that our approximation, performs well in practice (avg. error of < 10%). Note, our approximation concerns the *latency* introduced by LLC misses, *not the number* of LLC misses—the PIX back-end tracks LLC misses per path in Step 1.

Finally, PIX instantiates each formula in the general-case interface with these inferred distributions to compute the requested latency statistic (e.g., 50th percentile in Fig. 2). We show examples of deployment-specific interfaces and their distributions in §4.

---

[2]The operator cannot be certain that this general-case interface is accurate for the production binary, but we do not see this as a barrier to adoption: operators routinely deploy NF binaries while relying only on non-attested configuration interfaces and vendor manuals [55].

---

[3]The expert must do this once per data structure, like the pre-analysis.

Framework	NF	Functionality
eBPF XDP	Katran LB	Per-flow state, per-VIP state, consistent hashing, IPv6, ICMP, QUIC, tunneling
	Cilium filter	Longest prefix matching, IPV6
	CRAB LB	Read-only state
	hXDP firewall	Per-flow state
DPDK	Natasha NAT	Per-flow state, handles fragmentation, UDPLite, ICMP, ARP
	Maglev LB	Per-flow state, consistent hashing
	VigNAT	Per-flow state, header rewriting
	Bridge	Packet duplication
	Router	Longest prefix matching
	Policer	Per-flow state, fine-grained timing
	DPDK NAT	Per-flow state, header rewriting, cksum offload
	DPDK firewall	Per-flow state

Table 2. Network functions used to evaluate PIX.

4 Evaluation

In this section, we address two main questions: (1) does PIX extract good performance interfaces, and (2) can performance interfaces make NF developers and NF operators more productive? To answer the former, we quantitatively evaluate the complexity of PIX-extracted interfaces, their accuracy, and the time it takes to obtain them (§4.1). We find that they are one to two orders of magnitude simpler and more accurate than prior work. To answer the latter question, we show how developers can use PIX-extracted interfaces to catch performance regressions and fix performance bugs (§4.2). We then show how operators can use interfaces to pick the NF variant best suited for their target hardware and to perform root-cause diagnosis of performance anomalies (§4.3).

We evaluate PIX on 12 dataplane NFs that cover a wide variety of functionality and network protocols (Table 2). These include the Katran load balancer used in production at Facebook [71], the Natasha NAT used in production at Scaleway [58], the XDP packet filter from the Cilium project [14] and an implementation of Google's Maglev load balancing algorithm [25]. The NFs were written using DPDK [21] and eBPF XDP [82], arguably the two most popular frameworks today for building high-performance software NFs. VigNAT, Policer, Router and Bridge come from the Vigor project [83], the CRAB load balancer from [46], and the hXDP firewall from [8]. The Vigor and eBPF NFs are written in the commonly used stateless/stateful split model, which makes them amenable to exhaustive symbolic execution. We modified Natasha and DPDK NAT to also have such a clean split; this took ~3 person-days per NF.

The performance metrics we use for DPDK-based NFs are `x86` instruction count, `x86` memory access count, and `x86` CPU cycles (thus wall-clock time). Note, PIX is not specific to x86 and can just as easily predict the corresponding metrics for another ISA (e.g., ARM) if the PIX front-end is given the corresponding binary. For eBPF NFs, we only analyze the NF itself, and not the eBPF maps that are part of Linux, so we only report hardware-independent metrics.

NF	Implementation		HW-independent interface (PIX)		HW dependent interface (PIX)		Bolt contract	
	LOC	CC	LOC	CC	LOC	CC	LOC	CC
Natasha	2932	192	1.8%	8.9%	2.8%	15.1%	17.4%	97.3%
Maglev	3168	29	0.9%	37.9%	1.6%	65.5%	2.1%	82.7%
VigNAT	2770	22	0.7%	36.3%	0.9%	52%	1.8%	81%
Bridge	2837	219	0.5%	2.7%	2.1%	10.5%	22.7%	98.6%
Router	1260	17	0.4%	17.6%	1.0%	29.4%	3.0%	82.3%
Policer	2466	16	0.4%	31.2%	0.6%	37.5%	1.4%	81.2%
DPDK FW	2508	21	0.8%	38%	1.0%	45%	2.0%	85.7%
DPDK NAT	1780	35	0.6%	27%	0.9%	39%	4.5%	80%
Katran	2661	3226	2.8%	0.8%	-	-	363%	100%
Cilium filter	784	42	3.2%	14.3%	-	-	10.7%	100%
CRAB	437	4	2.0%	100%	-	-	2%	100%
hXDP FW	312	33	3.8%	15.1%	-	-	30%	100%

Table 3. Complexity of extracted interfaces and Bolt contracts vs NF implementation. "($x\%$)" means "x of implementation". For each NF, the complexity is calculated for an interface with resolution equal to 10% of the maximum latency variability the NF can exhibit. Since Bolt computes the worst-case performance for HW-dependent metrics (not shown), the numbers are identical to those for HW-independent metrics.

Our testbed consists of two directly connected servers: a device under test (DUT) and a traffic generator and sink (TG). The servers are identical, with an Intel Xeon E5-2667 v2 processor @ 3.30 GHz, 32 GB of DRAM, and Intel 82599ES 10-Gbps NICs. The DUT runs one of the NFs and measures the performance, while the TG uses MoonGen [26] to generate traffic.

4.1 Does PIX Work?

In this section, we show that extracted interfaces are $1-2$ orders of magnitude simpler than both the NF implementations and the equivalent Bolt contracts (§4.1.1). Their accuracy is 100% for reasonable resolutions, while at the finest resolution they are still practical and considerably better than Bolt, the state of the art (§4.1.2). Extracting a performance interface typically takes minutes (§4.1.3).

4.1.1 Are performance interfaces user-friendly?

To evaluate the "human palatability" of the performance interfaces, we (1) measure their complexity in terms of both lines of code (LOC) and cyclomatic complexity (CC) [78], and (2) evaluate whether the primitives exposed by the performance interfaces are those that NF developers and operators are familiar with.

Table 3 compares the complexity of the PIX-extracted interfaces and the Bolt contracts, measured as a fraction of LOC and CC of the implementation. In a nutshell, the extracted interfaces have $26-210\times$ fewer LOC than the corresponding implementations and are $3-124\times$ less cyclomatically complex, ignoring CRAB, which is already simple to start with. The performance resolution allows PIX-extracted interfaces to be $2.3-129\times$ shorter than the Bolt contracts, and $2.1-124\times$ less cyclomatically complex, by abstracting away irrelevant details. The more complex an NF, the higher this reduction in complexity,

which argues for the real-world utility of performance interfaces.

Fig. 5 illustrates the impact of varying resolution on the complexity of Katran's performance interface. At the finest granularity, Katran's instruction-count interface, like the Bolt contract, is fairly complex (LOC=9675, CC=3226 independent paths). Since no two packets in Katran can incur an instruction count that differs by more than 854 instructions (number determined by PIX and verified by us), for resolutions above 854 the interface becomes a simple upper bound. In between these two extremes, we see how low-level details get abstracted away—for instance, at resolution=50 instructions, we see a 125× drop in complexity (LOC=75, CC=26). The Bolt contract, however, lacks the notion of resolution and thus remains 3.6× longer and just as cyclomatically complex as the implementation.
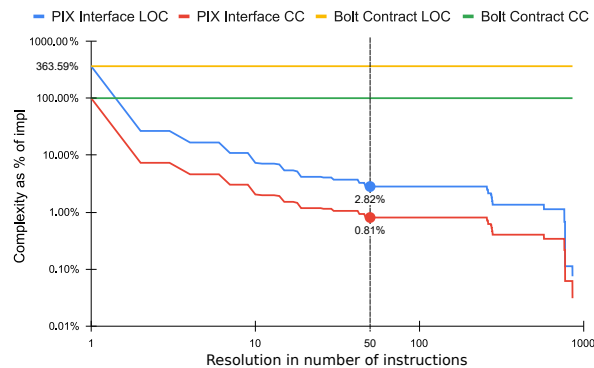

Figure 5. Impact of varying resolution on the size (LOC) and complexity (CC) of Katran's performance interface.

We conclude that PIX-extracted performance interfaces are significantly simpler than the NF implementations, which argues for them making it easier to understand performance behaviors by reading the interface than reading the code. The notion of resolution succeeds in abstracting a performance interface, giving the reader a knob with which to control the amount of detail contained in the interface.

Another aspect of palatability is how familiar the interface looks to a human reader. To illustrate this, we show an example of the general-case interface for VigNAT in Fig. 6, restricted to TCP/UDP packets for space considerations. The interface is a succinct, self-descriptive Python program. The conditions in `if` statements are expressed in terms of fields in the input packet header (e.g., `pkt.port`) or semantic operations on data structures (e.g., `nat_flowtable.contains`), which are primitives we expect both developers and operators to understand. Being a stateful NF, VigNAT's performance is influenced by NF state, and the interface reflects this via

```
def perf_vignat_gc(pkt):
  # Perf metric: x86 instructions
  # Resolution: 10
  # NF state:
  #   flowtable
  # PCVs:
  #   e - expired flows
  #   t - bucket_traversals
  #   c - hash_collisions

  x = 19*e*t + 40*e*c + 227*e + 123

  if not (pkt.is_IP) or not(pkt.is_TCP or pkt.
    is_UDP):
    return x + 7
  else:
    if pkt.port != internal_network_port:
      if flowtable.contains(pkt.flow):
        return x + 289
      else:
        return x + 68
    else:
      if flowtable.contains(pkt.flow):
        return x + 18*t + 30*c + 395
      else:
        return x + 31*t + 30*c + 547
```

Figure 6. Extracted general-case interface for VigNAT.

PCVs, documented in the header. Bolt, on the other hand, does not translate low-level details and exposes primitives such as the starting condition on line 4 of Fig. 4. While such details are understandable to the NF's developer, they make the contract hard to read for those unfamiliar with the code.

Finally, we illustrate the impact of deployment-specific instantiation of interfaces on their palatability. Fig. 7 shows the interfaces for VigNAT's 50th and 95th percentile latencies and the distribution underlying them, for a particular *<workload, HW>* pair. The deployment-specific instantiation turns each formula (expressed in terms of PCVs in the general-case interface) into concrete values specific to the environment and workload, thus tailoring the interface to an operator's needs. The latency CDF also enables interested operators to understand how VigNAT's percentile latency varies.

4.1.2 Accuracy of performance interfaces

We now evaluate the prediction error of PIX-extracted interfaces, i.e., the difference between the latency predicted by the interfaces and the measured latency.

To do so, we use PIX to extract interfaces for all 8 DPDK NFs[4] for two hardware-independent metrics (`x86` instructions and memops) and one hardware-dependent one (`x86` cycles). For each NF, we instantiate two deployment-specific interfaces corresponding to two very different deployments—typical traffic representative of university networks [6] and adversarial traffic that seeks denial-of-service [64]. The above deployments represent opposite ends of the spectrum for *absolute* NF latencies [64]— e.g., adversarial traffic incurs 2.1× greater latency than typical traffic in VigNAT. To instantiate each deployment-specific interface, we use PCAP traces of 100M packets each. These traces are similar to what an operator could

---

[4]PIX does not support HW-dependent metrics for eBPF NFs

```python
1  def perf_vignat_ds(pkt):
2    # Metric: CPU cycles, Resolution: 200
3    # Percentile: 50
4    # NF state - flowtable
5
6    if flowtable.contains(pkt.flow):
7      return 301
8    else:
9      if pkt.port != internal_network_port:
10       return 92
11     else:
12       return 558
```

```python
1  def perf_vignat_ds(pkt):
2    # Metric: CPU cycles, Resolution: 200
3    # Percentile: 95
4    # NF state - flowtable
5
6    if flowtable.contains(pkt.flow):
7      return 395
8    else:
9      if pkt.port != internal_network_port:
10       return 97
11     else:
12       return 1037
```
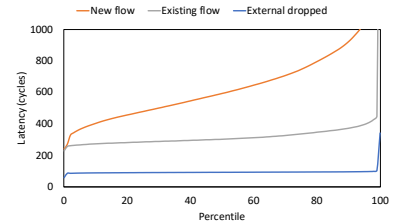
Figure 7. Common-case interfaces for VigNAT (50th and 95th percentile) and the latency CDF (resolution=200 cycles).

obtain with `tcpdump` on their domain gateway and are not specific to any particular NF implementation.

For ground-truth measurements, we manually generate synthetic packet traces for each *<NF, deployment>* pair akin to Scaleway's NAT test suite [57]. We playback these traces against the NF and measure the latency of each packet (the ground truth). Note, the synthetic traces are only used to measure the ground truth and not for predicting performance, thus avoiding any overfitting.

To compare to Bolt [41], the closest prior work, we use their published code [1]. We run each deployment trace through the Bolt distiller, which computes the PCVs and concretizes the performance contracts. For a comparison to Freud [69], please see Appendix A.

We present here the prediction error for the 50th percentile, 90th percentile and 99th percentile latencies (which is the point at which PIX's limitations become evident). Appendix B provides the details for the entire spectrum. We compute all prediction errors by subtracting the relevant statistic of the measured latency distribution from that of the predicted latency distribution. The results reported are at resolution 1, where PIX does the worst.

50th percentile (median) latency: Table 4 describes the maximum and average error for median latencies across all NFs for each metric and deployment regime. Note, despite the *absolute* NF latency differing widely, PIX's prediction accuracy is similar for both deployments, showing that the PIX front-end correctly instantiates each deployment-specific interface.

	Instr's error		**Memops** error		**Cycles** error	
	Typ	*Adv*	*Typ*	*Adv*	*Typ*	*Adv*
PIX	1.8% (1.5%)	1.7% (1.2%)	4% (1.6%)	3.7% (1.5%)	26% (11%)	24% (9%)
Bolt	7.5% (3.7%)	7.6% (4.0%)	7.5% (3.7%)	7.6% (4.0%)	308% (164%)	186% (103%)
PIX improvement	4.1× (2.4×)	4.4× (3.3×)	1.8× (2.3×)	2.0× (2.6×)	11.8× (14.9×)	7.7× (11.4×)

Table 4. Max (average) median latency prediction error for PIX and Bolt for typical (Typ) and adversarial (Adv) traffic.

We find that, even in the worst case for PIX (i.e., finest resolution), the error for hardware-independent metrics is ≤4%, which is small enough to make PIX practical.

At any reasonable resolution, the error vanishes and PIX becomes 100% accurate. PIX outperforms Bolt by 4.4×.

For the CPU cycles, PIX has a maximum error of 26%. This is due to the overhead of the instrumentation used to measure the CPI and LLC miss latencies. Nevertheless, PIX's accuracy is an order of magnitude better than Bolt's since PIX reasons about hardware performance as a distribution, while Bolt only models the worst case.

90th percentile latency: Table 5 describes the prediction error for 90th percentile latencies. The results are similar to those for median latency with PIX outperforming Bolt by up to an order of magnitude. This is once again due to PIX reasoning about each of the PCVs as a distribution, while Bolt only models the worst-case.

	Instr's error		**Memops** error		**Cycles** error	
	Typ	*Adv*	*Typ*	*Adv*	*Typ*	*Adv*
PIX	1.4% (0.9%)	1.2% (0.9%)	3.2% (1.4%)	2.8% (1.2%)	22% (10%)	19% (7%)
Bolt	5.9% (2.4%)	5.3% (3.1%)	6.6% (2.9%)	6.1% (3.2%)	234% (122%)	153% (94%)
PIX improvement	4.2× (2.6×)	4.4× (3.4×)	2.0× (2.1×)	2.1× (2.6×)	10.6× (12.2×)	8× (13.4×)

Table 5. Max (average) prediction error for 90th percentile latencies for typical (Typ) and adversarial (Adv) traffic.

99th percentile latency: PIX cannot accurately predict the latency at the very end of the tail (nor can Bolt). PIX's predictions have an error of ≤61% (average 22%), while Bolt's predictions have an error of ≤45% (average 14%).

It is interesting to note that PIX *underestimates* the 99th percentile latency while Bolt *overestimates* it; this contrasting behavior is due to the different hardware models underlying the two tools. PIX underestimates the 99th percentile latency since its simple hardware model (*instructions * CPI + LLC_misses * miss_latency*) is invalid at this percentile where other hardware aspects also impact latency significantly. Bolt, on the other hand, overestimates the 99th percentile latency since its hardware model is designed to estimate the absolute worst-case latency. However, PIX's simple hardware model enables it to accurately predict performance at all percentiles except the tail (details in Appendix B); a task that Bolt's worst-case-only model is incapable of.

4.1.3 Time to extract performance interfaces

Table 6 shows the time it takes PIX to extracts the general-case interfaces for all the NFs in this evaluation. We believe that these numbers make it feasible to incorporate performance interfaces extraction part of the regular NF development cycle, e.g., as part of continuous integration.

NF	Natasha	Maglev	VigNAT	Bridge	Router	Policer	DPDK FW	DPDK NAT	Katran	Cilium filter	CRAB	hXDP FW
PIX	15	5	4	17	0.73	3	4	6	32	0.43	0.15	0.23
BOLT	6	4	2	7	0.35	1.7	2	3	28	0.26	0.1	0.13

Table 6. Time, in minutes, for PIX and Bolt to extract the general-case interfaces and contracts, respectively.

The time required to obtain the deployment-specific interface is largely a function of the time required to run the provided workload. In our experiments, we ran PCAP files with 100M packets, and it took PIX <= 5 mins to generate the deployment-specific interface for a given *<workload, HW>* pair from the general-purpose interface, regardless of NF. We conclude that PIX fulfills the portability requirement (§3) well: operators can download an NF with its general-case interface, provide a PCAP file specific to their deployment, and PIX quickly produces the deployment-specific interface.

4.2 Are interfaces useful to NF developers?

In this section, we present two workflows that NF developers can use to understand (§4.2.1) and debug (§4.2.2) the performance behavior of their code.

4.2.1 Flagging performance regressions

Programmers often introduce involuntarily performance regressions. Using performance test suites to catch such regressions is not easy, because they require environment setup, are fragile, and take long to run. We show here how a developer or a tool can instead compare the performance interface before and after a commit to identify performance regressions more quickly, conveniently, and precisely than with a performance test suite.

We wrote a script that retrieves each Katran commit and uses PIX to extract the corresponding instruction-count interface, at resolution=1. For each pair of commits a and b, there is a corresponding pair of interfaces S_a and S_b. The script finds the maximum latency (in terms of LLVM instruction count) predicted by each of the two interfaces and compares the two. We report LLVM (not eBPF bytecode) instructions since PIX builds on KLEE which interprets LLVM IR. Reporting eBPF instructions would require us to build on a tool that interprets eBPF bytecode (e.g., Serval [59])—this is an engineering task

we leave to future work. We run PIX on all commits to the eBPF portion of Katran's code.

Table 7 shows the commits where a performance regression occurs. Over the past three years, the maximum latency for new flows regressed by 14.6%.

Commit ID	Perf before [LLVM instr's]	Perf after [LLVM instr's]	Performance regression [%]
Orig commit	-	1771	-
873d0501695c	1765	1896	7.42%
39e58b530a8a	1896	1914	0.95%
458aa0907b68	1914	1933	0.99%
15f81d0e7ec6	1930	1946	0.83%
74c3338c2f7e	1952	1983	1.59%
d0790d3a3823	1983	2030	2.37%
All commits	1771	2030	14.62%

Table 7. Perf regressions in Katran (handling new flows).

We imagine using this workflow as part of continuous integration (CI) to automatically identify unintended performance regressions. The CI system can present to the developer a before-and-after comparison of performance that directly highlights for which classes of inputs the regression occurs and what the magnitude of the regression is. Compared to performance tests, this workflow consumes less developer time and fewer resources and offers better completeness.

4.2.2 Fixing performance bugs

By helping developers understand the code's performance more quickly and deeply, interfaces can help fix performance bugs. We illustrate this with two examples of performance bugs in the map used by Vigor NFs [50].

The top of Fig. 8 shows a snippet of the performance interface of the `contains` operation in libVig's `map`.

```
if map.contains(key): # --- BEFORE ---
  if not(cached(key)):
    # Warning: 2*t integer divides
    return (4*t)*miss_latency + (21*t+27)*CPI
....
if map.contains(key): # --- AFTER ---
  if not(cached(key)):
    return (1*t)*miss_latency + (18*t+27)*CPI
....
```

Figure 8. Interface for `map_contains()` before and after the bug fix. `t` is the PCV for traversals in the hash ring.

The first red flag is the warning issued by PIX itself, based on tracking of expensive x86 instructions that adversely impact CPI. Looking for integer divides in the `map` code, we found that, on each traversal, it uses two costly modulo operations. To fix the issue, we replaced them with one bitwise `and`.

The second red flag is that each traversal requires 4 independent heap accesses (`4*t`). It turns out that `key` metadata is being stored in four distinct arrays of `int`

elements. Our fix was to encapsulate `key`'s metadata in a single `struct` and use a single array with elements of this `struct` type. The rest remained unchanged.

Table 8 shows the impact of our fixes, based on Vigor's benchmarks: the two fixes, together, improve NF latency by 22% on average, and throughput by 19%.

NF	Throughput [Mpps]				Latency [ns]			
	Orig	Fix 1	Fix 2	**Change**	Orig	Fix 1	Fix 2	**Change**
VigNAT	3.88	4.36	4.68	20.62%	317	276	236	25.55%
Bridge	3.05	3.59	3.62	18.69%	410	332	323	21.22%
Maglev	2.58	2.86	3.04	17.83%	482	423	391	18.88%

Table 8. Throughput and latency of three NFs using `map`, shown before/after each performance bug fix.

4.3 Are interfaces useful to NF operators?

Operators typically care about how an NF performs in their specific deployment, not in general for everyone's deployment. We show how operators can use performance interfaces to pick the NF variant best suited to their hardware (§4.3.1) and to do a root-cause diagnosis of deployment-specific performance anomalies (§4.3.2).

4.3.1 Which NF variant for my NIC?

Modern NICs provide the ability to offload specific tasks (like checksums and encryption) to specialized hardware. It is therefore useful to know which variant of an NF takes max advantage of the offloads available on a NIC.

Fig. 9 shows the interfaces for two variants of a NAT, and the interaction with checksum offload on Mellanox ConnectX-4 [15] and Intel Ixgbe [40] NICs. The formally verified VigNAT does not do any offloading, whereas DPDK NAT does. The strings in the `if` conditions on lines 3 and 6 are identical to the one used by the NIC driver to identify itself [22]. The interface also shows the difference in latency: Ixgbe requires the software to compute a pseudo-header checksum, whereas ConnectX-4 allows full offload, so it has lower latency.

Based on this performance interface, an operator can make an informed deployment decision: if using Ixgbe NICs, choosing the verified VigNAT makes sense; else, it's a trade-off to make carefully.

4.3.2 Why do I get bad performance?

NFs running in production can face workloads that trigger surprising performance degradation. To address such anomalies, operators must first diagnose the root cause, and this often takes a lot of work.

PIX helps the search for a root cause by providing a list of possible explanations for the observed performance, ranked by likelihood. Given a problematic workload and an NF (or its general-case interface), PIX instantiates the PCVs in a deployment-specific manner and then measures the distributions for each PCV and the NF

```
# Snippet from VigNAT interface
if flowtable.contains(pkt.flow):
  return  18*t + 30*c + 518  # No offload
else:
  ....
# Snippet from DPDK NAT interface
if flowtable.contains(pkt.flow):
  if(NIC_family == "net_mlx5"):
    return  18*t + 30*c + 265 + cksum_offload()
  else:
    if(NIC_family == "net_ixgbe"):
      return 18*t + 30*c + 478 + cksum_offload()
    else:
      return 18*t + 30*c + 564
else:
  ....
```

Figure 9. Interfaces for VigNAT and DPDK NAT: VigNAT does checksums in software, while DPDK NAT offloads checksums to the NIC as much as possible.

latency. It then ranks the PCVs based on the correlation between the latency distribution and that of the PCV (using least-square fit linear regression).

To illustrate this workflow, we refer to three performance bugs that span both hardware and software root causes, shown in Table 9. The first bug occurs due to the uniform random workload causing hash collisions in a widely used hash function [42] used by Bridge; typical workloads with Zipfian distributions do not suffer from hash collisions. The second bug is caused by VigNAT's batches expiry of flows, which results in a latency spike that only becomes evident for traffic with high churn. The third bug occurs when the active flowtable in Maglev overflows the last-level cache of the server; this makes the latency spike be highly dependent on LLC configuration.

Bug	Root cause	Identified as most-likely cause?
Spike in median latency of Bridge for uniform random workload	`hash-collisions`	Yes
Spike in tail latency of VigNAT due to high churn	`expired-flows (batched)`	Yes
Spike in median latency of Maglev on a particular x86 server	`active-flowtable-size`	Yes

Table 9. Performance bugs used for root-cause diagnosis.

For each bug, we generated a workload that triggers it and provided the PCAP file to PIX, along with the general-case interface of the corresponding NF. For each bug, PIX correctly reported the culprit PCV as the most likely root cause. Of course, PIX can only track bugs that arise from PCVs it accounts for. It would be unable, for instance, to identify the root cause for a latency spike due to LLC evictions caused by a noisy neighboring process, since PIX does not account for contention.

This example illustrates how PIX can help focus the operators' attention on likely explanations for the performance they observe, thereby reducing the amount of work needed to find the root cause.

In conclusion, our evaluation shows that PIX is practical: the complexity of extracted interfaces is significantly lower than the NF implementation, their accuracy is high, and the time taken to extract them is reasonable. Further, NF developers and operators can use these interfaces to identify performance regressions, diagnose and fix performance bugs, and pick the NFs that are best suited to their hardware.

5   Does PIX Generalize?

In this section, we explore how PIX can generalize in two directions: (1) programs other than NFs, that are nevertheless still amenable to ESE; and (2) NFs that are not amenable to ESE. Overall, we find that the design of PIX—split into a modular back-end and front-end that produce general-case and deployment-specific interfaces, respectively—enables generalization by adapting just the necessary modules in the PIX pipeline.

Beyond NFs: We have successfully applied PIX to the OpenSSL library, to uncover digital side-channels, and to eBPF extensions for user-space file systems.

Extracting interfaces for finding digital side-channels required modifying only PIX's hardware model (i.e., step 1 in the back-end). Implementing a new model focused on sources of constant-time violations (using the exhaustive list in [3]) took us 2 person months. We ran PIX on 12 cryptographic primitives from OpenSSL 3.0 [61] and found a constant-time violation in the AES cipher unpadding function. This violation was acknowledged by the OpenSSL maintainers [62]. We have submitted a pull request [63] that has undergone multiple rounds of review and is in the final stages of getting merged.

Our experience with OpenSSL reinforced our belief (from §4.2.1) that a tool that automatically extracts performance interfaces would be of great use to developers. For example, we learned that the violation we uncovered had been latent since OpenSSL 1.1.1 because the developer "just reused the code" and had somehow been missed despite the extremely thorough code reviews that OpenSSL goes through. If performance interfaces of the OpenSSL code were extracted regularly, e.g., as part of continuous integration, it is unlikely that this violation would have persisted for this long.

Extracting interfaces for eBPF file system extensions was more straightforward since the code is similar to that of eBPF NFs. Here, we only had to add translation rules (step 2 in the PIX back-end) corresponding to the supported system calls. This took 4 person-days, after which PIX was able to automatically extract interfaces for the extFUSE extensions [7].

Code not amenable to ESE: To evaluate the limits of PIX's ESE-based approach, we used PIX on Snort [73],

a popular IDS that independent prior work has shown to not be amenable to ESE [53, 81]. Our results corroborated those from prior work; while PIX did extract performance interfaces for the networking stack and all detection rules that look only at packet headers, attempting to extract a complete interface caused PIX to time out. Extracting an interface from Snort with PIX requires either that we modify its code to cleanly separate the stateful components, or that we replace Bolt in the PIX back-end with a manual theorem prover.

6   Related Work

We compared PIX to Bolt [41] w.r.t the design (in §2) and results (in §4.1). We do not do so again here.

Here we provide a qualitative comparison of PIX against Freud, Appendix A provides a detailed quantitative one. Freud treats code almost as a black box and relies on developers to provide comprehensive performance test suites in order to guide the exploration of performance behaviors and ensure prediction accuracy. PIX is white-box because it analyzes source code. Analyzing the source ensures that PIX can analyze the system once, and instantiate the interface for different deployments, while Freud users must re-run the tool for each new <workload, hardware> pair. Lastly, Freud's performance formulae are limited to being expressed in terms of program variables, but the performance of stateful code typically depends on (implicit) PCVs instead [37, 38].

Performance upper bounds and adversarial workloads: Worst-Case Execution Time (WCET) Analysis derives formal upper bounds on performance; [79] provides an overview of the state of the art. These bounds are particularly popular in the domain of real-time and safety-critical systems where performance guarantees are a part of functional correctness. While WCET only looks at one aspect of the performance profile—the absolute worst-case—performance interfaces characterize performance in the face of any arbitrary input, whether typical, ideal, or adversarial. Further, to enable stringent upper bounds, real-time systems tend to avoid dynamic data structures and input-dependent memory accesses—aspects that are commonplace in NFs.

Considerable prior work focuses on generating and analyzing adversarial workloads that attack software performance [2, 18, 49, 60, 64, 65, 67, 72, 76]. As with WCET, all of this work focuses only on worst-case inputs, while interfaces reflect the entire performance profile.

Performance profilers: Traditional profilers [51] measure the execution cost (e.g., running time, executed instructions, cache misses) of a piece of code. Trend Profiling [34], Algorithmic Profiling [85] and Input-Sensitive Profiling [16, 17] take this one step further: by extracting a cost function defining the relationship between input

size and execution cost. However, like Freud, these tools treat the code as a black-box and require developers to provide comprehensive performance test suites to guide the exploration of performance behavior.

Performance analysis for SmartNIC-based NFs: Krude et al. [47] use SMT solvers to analyze NF code written for processor-based SmartNICs and provide lower bounds on throughput. Focussing solely on throughput lower bounds results in their approach being limited to analyzing worst-case latency, much like WCET. Clara [68] uses machine learning to analyze NF code written in C to identify "effective porting strategies" that result in low latency when the NF is ported to a SmartNIC. Unlike PIX that focusses on accurately predicting the NF latency, Clara focusses on identifying how the NF implementation can make best use of the SmartNIC hardware (e.g., accelerator usage, NF state placement strategies, etc).

Program analysis for NF code running on commodity hardware: Several instances of prior work have proposed using program analysis to help understand, debug, and verify the semantic behavior of software NFs [10, 11, 19, 44, 66, 74, 84, 86]. PIX builds upon the experience of all of this prior work, but analyzes NF performance.

NF performance monitoring and diagnosis: Several instances of prior work [28, 35, 56, 80] diagnose performance issues such as packet drops or low throughput in NF deployments. Such work is complementary to PIX since it helps diagnose performance issues once they occur in production, while PIX provides a summary of NF performance before the NF is deployed.

7 Conclusion

We proposed the notion of a *performance interface*—a program that accepts the same inputs as the system and outputs the latency incurred by the given input. For the interface to be simultaneously simple, accurate and human-readable we proposed (a) the notion of a *performance resolution* to eliminating unnecessary details, and (b) separate *deployment-specific interfaces* to tailor the interface to particular <workload, environment> pairs.

We described a tool (PIX) that automatically extracts performance interfaces from NF implementations. and evaluated it on 12 NFs, including several used in production. Our results show that PIX is practical—the complexity of extracted interfaces is significantly lower than the NF implementation, their accuracy is high, and the time to extract them is reasonable. Finally, we show how NF developers and operators can use these interfaces today, to identify performance regressions, diagnose and fix performance bugs, and pick the NFs that are best suited to their hardware.

8 Acknowledgements

References

[1] Bolt source code. https://github.com/bolt-perf-contracts/bolt.

[2] AFEK, Y., BREMLER-BARR, A., HARCHOL, Y., HAY, D., AND KORAL, Y. Making DPI engines resilient to algorithmic complexity attacks. *IEEE/ACM Trans. on Networking* (2016).

[3] ALMEIDA, J. B., BARBOSA, M., BARTHE, G., DUPRESSOIR, F., AND EMMI, M. Verifying constant-time implementations. In *USENIX Security Symp.* (2016).

[4] WSJ: Facebook, google and apple hit by unusual outages. https://www.wsj.com/articles/facebook-and-instagram-suffer-lengthy-outages-11552539752.

[5] BARBETTE, T., SOLDANI, C., AND MATHY, L. Fast userspace packet processing. In *ACM/IEEE Symp. on Architectures for Networking and Communications Systems* (2015).

[6] BENSON, T., AKELLA, A., AND MALTZ, D. A. Network traffic characteristics of data centers in the wild. In *Internet Measurement Conf.* (2010).

[7] BIJLANI, A., AND RAMACHANDRAN, U. Extension framework for file systems in user space. In *USENIX Annual Technical Conf.* (2019).

[8] BRUNELLA, M. S., BELOCCHI, G., BONOLA, M., PONTARELLI, S., SIRACUSANO, G., BIANCHI, G., CAMMARANO, A., PALUMBO, A., PETRUCCI, L., AND BIFULCO, R. hxdp: Efficient software packet processing on FPGA NICs. In *Symp. on Operating Sys. Design and Implem.* (2020).

[9] CADAR, C., DUNBAR, D., AND ENGLER, D. R. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symp. on Operating Sys. Design and Implem.* (2008).

[10] CANINI, M., KOSTIC, D., REXFORD, J., AND VENZANO, D. Automating the testing of OpenFlow applications. *Intl. Workshop on Rigorous Protocol Engineering* (2011).

[11] CANINI, M., VENZANO, D., PEREŠÍNI, P., KOSTIĆ, D., AND REXFORD, J. A NICE way to test openflow applications. In *Symp. on Networked Systems Design and Implem.* (2012).

[12] CHEN, S., DELIMITROU, C., AND MARTINEZ, J. F. PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems* (2019).

[13] Commits to the eBPF code in the Cilium project. https://github.com/cilium/cilium/commits/master/bpf.

[14] Cilium Project. https://cilium.io.

[15] Mellanox ConnectX-4 Network Adapter Cards. https://downloadcenter.intel.com/download/14687.

[16] COPPA, E., DEMETRESCU, C., AND FINOCCHI, I. Input-sensitive profiling. In *Intl. Conf. on Programming Language Design and Implem.* (2012).

[17] COPPA, E., DEMETRESCU, C., FINOCCHI, I., AND MAROTTA, R. Estimating the empirical cost function of routines with dynamic workloads. In *Intl. Symp. on Code Generation and Optimization* (2014).

[18] CROSBY, S. A., AND WALLACH, D. S. Denial of service via algorithmic complexity attacks. In *USENIX Security Symp.* (2003).

[19] DOBRESCU, M., AND ARGYRAKI, K. Software dataplane verification. In *Symp. on Networked Systems Design and Implem.* (2014).

[20] DOBRESCU, M., ARGYRAKI, K., AND RATNASAMY, S. Toward predictable performance in software packet-processing platforms. In *Symp. on Networked Systems Design and Implem.* (2012).

[21] DPDK: Data plane development kit. https://dpdk.org.

[22] Ethtool Driver Identifier. https://docs.huihoo.com/doxygen/linux/kernel/3.7/include$_2$uapi$_2$linux$_2$ethtool$_8$h$_s$ource.html#l00085.

[23] Commits to eBPF maps in the Linux Kernel. https://github.com/torvalds/linux/commits/master/kernel/bpf.

[24] eBPF maps. https://prototype-kernel.readthedocs.io/en/latest/bpf/ebpf$_m$aps.html.

[25] EISENBUD, D. E., YI, C., CONTAVALLI, C., SMITH, C., KONONOV, R., MANN-HIELSCHER, E., CILINGIROGLU, A., CHEYNEY, B., SHANG, W., AND HOSEIN, J. D. Maglev: A fast and reliable software network load balancer. In *Symp. on Networked Systems Design and Implem.* (2016).

[26] EMMERICH, P., GALLENMÜLLER, S., RAUMER, D., WOHLFART, F., AND CARLE, G. MoonGen: A scriptable high-speed packet generator. In *Internet Measurement Conf.* (2015).

[27] EYERMAN, S., EECKHOUT, L., KARKHANIS, T., AND SMITH, J. E. A performance counter architecture for computing accurate CPI components. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems* (2006).

[28] FAYAZBAKHSH, S. K., CHIANG, L., SEKAR, V., YU, M., AND MOGUL, J. C. Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags. In *Symp. on Networked Systems Design and Implem.* (2014).

[29] FILLIÂTRE, J., GONDELMAN, L., AND PASKEVICH, A. The spirit of ghost code.

[30] Freud source code repository. https://github.com/usi-systems/freud.

[31] FRIED, J., RUAN, Z., OUSTERHOUT, A., AND BELAY, A. Caladan: Mitigating interference at microsecond timescales. In *Symp. on Operating Sys. Design and Implem.* (2020).

[32] Ghost variables in software verification. http://whiley.org/2014/06/20/understanding-ghost-variables-in-software-verification/.

[33] GITHUB. The 2020 state of the Octoverse. https://octoverse.github.com, Dec. 2020.

[34] GOLDSMITH, S., AIKEN, A., AND WILKERSON, D. S. Measuring empirical computational complexity. In *Symp. on the Foundations of Software Eng.* (2007).

[35] GONG, J., LI, Y., ANWER, B., SHAIKH, A., AND YU, M. Microscope: Queue-based performance diagnosis for network functions. In *ACM SIGCOMM Conf.* (2020).

[36] Google cloud storage incident. https://status.cloud.google.com/incident/storage/19002.

[37] GULWANI, S. SPEED: symbolic complexity bound analysis. In *Intl. Conf. on Computer Aided Verification* (2009).

[38] GULWANI, S., MEHRA, K. K., AND CHILIMBI, T. M. SPEED: precise and efficient static estimation of program computational complexity. In *Symp. on Principles of Programming Languages* (2009).

[39] GUNAWI, H. S., HAO, M., LEESATAPORNWONGSA, T., PATANA-ANAKE, T., DO, T., ADITYATAMA, J., ELIAZAR, K. J., LAKSONO, A., LUKMAN, J. F., MARTIN, V., AND SATRIA, A. D. What bugs live in the cloud? A study of 3000+ issues in cloud systems. In *Symp. on Cloud Computing* (2014).

[40] Intel Network AdIntel 82599 10 GbE Controller Datasheetapter Driver for PCIe Intel 10 Gigabit Ethernet Network Connections. https://downloadcenter.intel.com/download/14687.

[41] IYER, R., PEDROSA, L., ZAOSTROVNYKH, A., PIRELLI, S., ARGYRAKI, K., AND CANDEA, G. Performance contracts for software network functions. In *Symp. on Networked Systems Design and Implem.* (2019).

[42] Java String hashCode. https://docs.oracle.com/javase/6/docs/api/java/lang/String.html#hashCode().

[43] JIN, G., SONG, L., SHI, X., SCHERPELZ, J., AND LU, S. Understanding and detecting real-world performance bugs. In *Intl. Conf. on Programming Language Design and Implem.* (2012).

[44] KHALID, J., GEMBER-JACOBSON, A., MICHAEL, R., ABHASHKUMAR, A., AND AKELLA, A. Paving the way for NFV: Simplifying middlebox modifications using statealyzr. In *Symp. on Networked Systems Design and Implem.* (2016).

[45] KING, J. C. Symbolic Execution and Program Testing. *J. ACM 19*, 7 (1976).

[46] KOGIAS, M., IYER, R., AND BUGNION, E. Bypassing the load balancer without regrets. In *Symp. on Cloud Computing* (2020).

[47] KRUDE, J., RÜTH, J., SCHEMMEL, D., RATH, F., FOLBORT, I., AND WEHRLE, K. Determination of throughput guarantees for processor-based smartnics. In *Intl. Conf. on Emerging Networking Experiments and Technologies* (2021).

[48] LAMPSON, B. Hints and principles for computer system design. https://www.microsoft.com/en-us/research/uploads/prod/2019/09/Hints-and-Principles-v1-full.pdf, November 2020.

[49] LEMIEUX, C., PADHYE, R., SEN, K., AND SONG, D. Perffuzz: automatically generating pathological inputs. In *Intl. Symp. on Software Testing and Analysis* (2018).

[50] libVig source code . https://github.com/vigor-nf/vigor/tree/master/libvig/verified.

[51] The Linux Perf Tool. https://en.wikipedia.org/wiki/Perf$_($Linux).

[52] MANOUSIS, A., SHARMA, R. A., SEKAR, V., AND SHERRY, J. Contention-aware performance prediction for virtualized network functions. In *ACM SIGCOMM Conf.* (2020).

[53] MEHROTRA, P., AND GOSWAMI, S. Analyzing Snort. Tech. rep., University of British Columbia, 2018.

[54] Microscope survey form and results. https://www.dropbox.com/s/66cp4k3wl8zm0q5/survey.pdf?dl=0.

[55] MOON, S., HELT, J., YUAN, Y., BIERI, Y., BANERJEE, S., SEKAR, V., WU, W., YANNAKAKIS, M., AND ZHANG, Y. Alembic: Automated model inference for stateful network functions. In *Symp. on Networked Systems Design and Implem.* (2019).

[56] NAIK, P., SHAW, D. K., AND VUTUKURU, M. NFVPerf: Online performance monitoring and bottleneck detection for NFV. In *IEEE Conf. on Network Function Virtualization and Software Defined Networks* (2016).

[57] Performance Tests for Natasha. https://github.com/scaleway/natasha/tree/master/test/perf.

[58] Scaleway Natasha. https://github.com/scaleway/natasha.

[59] NELSON, L., BORNHOLT, J., GU, R., BAUMANN, A., TORLAK, E., AND WANG, X. Scaling symbolic evaluation for automated verification of systems code with Serval. In *Symp. on Operating Systems Principles* (2019).

[60] OLIVO, O., DILLIG, I., AND LIN, C. Detecting and exploiting second order denial-of-service vulnerabilities in web applications.

In *Conf. on Computer and Communication Security* (2015).

[61] OpenSSL. https://github.com/openssl/openssl.

[62] Github issue raising constant-time violation in OpenSSL's Cipherblock Unpadding. https://github.com/openssl/openssl/issues/16230.

[63] Pull request to fix constant-time violation in OpenSSL's Cipherblock Unpadding. https://github.com/openssl/openssl/pull/16323.

[64] PEDROSA, L., IYER, R., ZAOSTROVNYKH, A., FIETZ, J., AND ARGYRAKI, K. Automated synthesis of adversarial workloads for network functions. In *ACM SIGCOMM Conf.* (2018).

[65] PETSIOS, T., ZHAO, J., KEROMYTIS, A. D., AND JANA, S. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Conf. on Computer and Communication Security* (2017).

[66] PIRELLI, S., VALENTUKONYTĖ, A., ARGYRAKI, K., AND CANDEA, G. Automated verification of network function binaries. In *Symp. on Networked Systems Design and Implem.* (2022).

[67] PUSCHNER, P., AND NOSSAL, R. Testing the results of static worst-case execution-time analysis. In *Real-Time Systems Symp.* (1998).

[68] QIU, Y., XING, J., HSU, K.-F., KANG, Q., LIU, M., NARAYANA, S., AND CHEN, A. Automated smartnic offloading insights for network functions. In *Symp. on Operating Systems Principles* (2021).

[69] ROGORA, D., CARZANIGA, A., DIWAN, A., HAUSWIRTH, M., AND SOULÉ, R. Analyzing system performance with probabilistic performance annotations. In *ACM EuroSys European Conf. on Computer Systems* (2020).

[70] SCHWARTZ, E. J., AVGERINOS, T., AND BRUMLEY, D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE Symp. on Security and Privacy* (2010).

[71] SHIROKOV, N., AND DASINENI, R. Open-sourcing Katran, a scalable network load balancer. https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer, May 2018.

[72] SMITH, R., ESTAN, C., AND JHA, S. Backtracking algorithmic complexity attacks against a NIDS. In *Annual Computer Security Applications Conf.* (2006).

[73] Snort. https://www.snort.org.

[74] STOENESCU, R., POPOVICI, M., NEGREANU, L., AND RAICIU, C. Symnet: Scalable symbolic execution for modern networks. In *ACM SIGCOMM Conf.* (2016).

[75] TERPSTRA, D., JAGODE, H., YOU, H., AND DONGARRA, J. J. Collecting performance data with PAPI-C. In *Workshop on Parallel Tools for High Performance Computing* (2009).

[76] TOFFOLA, L. D., PRADEL, M., AND GROSS, T. R. Synthesizing programs that expose performance bottlenecks. In *Intl. Symp. on Code Generation and Optimization* (2018).

[77] TOOTOONCHIAN, A., PANDA, A., LAN, C., WALLS, M., ARGYRAKI, K. J., RATNASAMY, S., AND SHENKER, S. Resq: Enabling slos in network function virtualization. In *Symp. on Networked Systems Design and Implem.* (2018).

[78] WATSON, A. H., AND MCCABE, T. J. *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric.* Computer Systems Laboratory, National Institute of Standards and Technology, 1996.

[79] WILHELM, R., ENGBLOM, J., ERMEDAHL, A., HOLSTI, N., THESING, S., WHALLEY, D., BERNAT, G., FERDINAND, C., HECKMANN, R., MITRA, T., MUELLER, F., PUAUT, I., PUSCHNER, P., STASCHULAT, J., AND STENSTRÖM, P. The worst-case execution-time problem — overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.* (2008).

[80] WU, W., HE, K., AND AKELLA, A. PerfSight: Performance diagnosis for software dataplanes. In *Internet Measurement Conf.* (2015).

[81] WU, W., ZHANG, Y., AND BANERJEE, S. Automatic synthesis of nf models by program analysis. In *ACM Workshop on Hot Topics in Networks* (2016).

[82] Express data path. https://en.wikipedia.org/wiki/Express_Data_Path.

[83] ZAOSTROVNYKH, A., PIRELLI, S., IYER, R. R., RIZZO, M., PEDROSA, L., ARGYRAKI, K. J., AND CANDEA, G. Verifying software network functions with no verification expertise. In *Symp. on Operating Systems Principles* (2019).

[84] ZAOSTROVNYKH, A., PIRELLI, S., PEDROSA, L., ARGYRAKI, K., AND CANDEA, G. A formally verified NAT. In *ACM SIGCOMM Conf.* (2017).

[85] ZAPARANUKS, D., AND HAUSWIRTH, M. Algorithmic profiling. In *Intl. Conf. on Programming Language Design and Implem.* (2012).

[86] ZENG, H., KAZEMIAN, P., VARGHESE, G., AND MCKEOWN, N. Automatic test packet generation. In *Intl. Conf. on Emerging Networking Experiments and Technologies* (2012).

Appendix A   Using Freud on NFs

In this section, we describe our experience experimenting with Freud. We used the publicly available Freud code [30] at commit ID e6e7a91006.

Freud takes as input a binary and a test suite, and outputs an expression of performance (runtime) as a function of input and global variables. So, by design, it strikes a different generality/accuracy balance than PIX: It is more general, in the sense that it can run on any program—not just NFs that are amenable to ESE—and requires no source code and no data-structure pre-analysis. It is less accurate, in two ways: (a) It cannot reason about the performance of execution paths that are not triggered by the test suite (since it does not symbex the program, and it does not analyze the source code). (b) It cannot reason about how past inputs affect performance in stateful code (since it does not know anything about the data structures where the state is stored).

To assess Freud's generality/accuracy balance, especially in the context of NFs, we used it on three classes of programs: (a) A stateless program that spins for a period of time proportional to the input length. (b) Data structures commonly used by NFs: a longest prefix match (LPM) trie and a hash map. (c) NFs: VigNAT (academic prototype), Natasha (production NAT used at ScaleWay), and Maglev (DPDK implementation of Google's load balancer). Natasha comes with an open-source performance test suite [57], making it an ideal fit for Freud. For the remaining programs, we used as test suites the packet traces on which we evaluated PIX.

Table 10 summarizes our results, discussed below.

Freud mode	Program	Accurate annotation?
Freud-vanilla	Synthetic stateless NF	Yes
	LPM trie	No
	Hashmap	No
	Real NFs	No
Freud-nf	Synthetic stateless NF	Yes
	LPM trie	Yes
	Hashmap	No
	Real NFs	No

Table 10. Summary of our experiments with Freud.

Freud-vanilla: First, we ran Freud on unmodified programs, and it behaved as expected: It successfully characterized the spinning program's runtime as a function of the input length, but it could not produce meaningful performance annotations for the data structures or NFs. This is normal, since, in the latter programs, runtime is a function of implicit variables that capture the interaction between current and past inputs (e.g., number of iterations of `while(bucket[i].is_full ==1)`).

Freud-nf: Next, to compare with PIX more fairly, we explicitly modified our programs to work with Freud: we identified conditions that we knew impacted performance (essentially PCVs) and manually added them as global variables (which Freud tracks). For instance, in the hashmap, we added a global variable to explicitly track the number of collisions; in the LPM trie, we added a global variable to explicitly track the depth traversed.

The results for the data structures were mixed: For the LPM trie, Freud produced an accurate performance annotation. For the hashmap, Freud mistook a correlation for a causation: when a test caused every packet to experience a collision, Freud concluded that runtime was determined by occupancy, as opposed to the number of collisions. We expect that this issue can be resolved at the cost of extra developer effort (to produce a smarter test suite).

For the real NFs, Freud could not produce meaningful performance annotations (despite our modifications to the NF source code). This is not surprising, given that Freud does not analyze the source code, hence is unable to track how a sequence of state-accessing calls affects runtime. For instance, in Maglev, known client packets that are destined to a now-stale backend-server undergo consistent hashing once again, to pick a new backend. Since Freud does not analyze the source code, it cannot track how this call sequence affects runtime, looking instead to express runtime as a function of individual variables—which does not work. We observed similar scenarios in the other NFs.

Conclusion: In its current form, Freud cannot produce accurate performance annotations for stateful NFs. To do so, it would need to track how a sequence of state-accessing calls affects performance. We think that that would necessarily require (a) some assumption about the

structure of the code (akin to our clean state assumption), (b) a nuanced test suite for the NF's data structures to reveal which aspects of state affect performance (which is done, in our approach, with the manual extraction of PCVs during pre-analysis), and (c) leveraging call context. We think that adding these elements to Freud would bring it very close to PIX; we expect it would achieve similar accuracy, but at the cost of its current generality.

Appendix B   Accuracy of performance interfaces

This section provides more detailed answers to the following questions: (1) What is the prediction error for both PIX and Bolt for each individual NF? (2) What is the prediction error for both PIX and Bolt as a function of the percentile latency?

Prediction error for individual NFs: Table 11 provides detailed per-NF results for PIX's prediction accuracy for hardware-independent metrics, i.e., x86 instruction count and x86 memory accesses. We see that the results are similar across all the NFs, with PIX consistently outperforming Bolt.

NF	Spade Prediction Error		Bolt Prediction Error	
	x86 instructions	x86 mem-ops	x86 instructions	x86 mem-ops
VigNAT	1.2%	1.3%	3.1%	4.0%
Bridge	0.8%	1.1%	3.6%	3.6%
Maglev	1.1%	1.1%	5.1%	4.2%
Router	1.7%	3.9%	6.8%	6.1%
Policer	1.4%	1.7%	4.2%	5.1%
Natasha	2.6%	3.2%	5.1%	5.6%
DPDK NAT	0.9%	1.1%	2.3%	2.9%
DPDK FW	1.1%	1.4%	2.7%	3.7%

Table 11. Prediction error for median latency for x86 instruction count and memory accesses for all 8 DPDK NFs in the deployment characterized by the typical traffic. The numbers for adversarial traffic are similar.
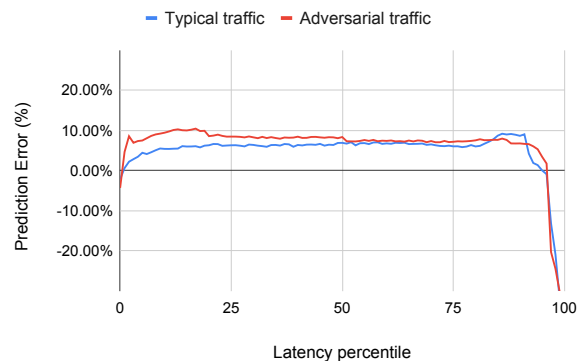

Figure 10. PIX's average prediction error for CPU cycles across two deployments as a function of the percentile latency
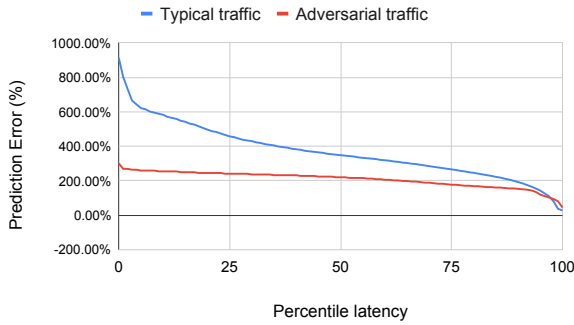
Figure 11. Bolt's average prediction error for CPU cycles across two deployments as a function of the percentile latency

Prediction error as a function of the percentile latency: Fig. 10 illustrates PIX's average prediction error across all 8 NFs for each deployment. First, we see that the average error shows similar trends across deployments, proving that PIX characterises the deployment-specific workload correctly. Second, we see that for both deployments the average error is more or less stable at around 8% up to the 95th percentile showing that for these percentiles, PIX characterises the interaction of the NF with the hardware correctly. Lastly, we see that at the tail, the prediction errors become negative. This is due to the fact that the simple HW model that PIX employs (*instructions * CPI + LLC_misses * miss_latency*) is invalid at the tail, where other hardware aspects kick in.

Fig. 11 illustrates Bolt's average prediction error across all 8 NFs for each deployment. Bolt estimates only worst-case latency and this is evident in the results—note the change in scale on the y-axis from Fig. 10. For all percentiles except the tail, Bolt is widly inaccurate with errors up to 900%. On the other hand, Bolt does not underestimate latency at the tail since it accounts for myriad worst-case scenarios in the underlying hardware worst-cases that PIX ignores to ensure accuracy across the remainder of the spectrum.

Automated Verification of Network Function Binaries

Solal Pirelli[+]   Akvilė Valentukonytė[°*]   Katerina Argyraki[+]   George Candea[+]

[+]EPFL   [°]Citrix Systems

Abstract

Formally verifying the correctness of software network functions (NFs) is necessary for network reliability, yet existing techniques require full source code and mandate the use of specific data structures.

We describe an automated technique to verify NF binaries, making verification usable by network operators even on proprietary code. To solve the key challenge of bridging the abstraction levels of NF implementations and specifications without special-casing a set of data structures, we observe that data structures used by NFs can be modeled as maps, and introduce a universal type to specify both NFs and their data structures, the "ghost map". In addition, we observe that the interactions between an NF and its environment are sufficient to infer control flow and types, removing the requirement for source code.

We implement our technique in KLINT, a tool with which we verify, in minutes, that 7 NF binaries satisfy their specifications, without limiting developers' choices of data structures. The specifications are written in Python and use maps to model state. KLINT can also verify an entire NF binary stack, all the way down to the NIC driver, using a minimal operating system. Operators can thus verify NF binaries, without source code or debug symbols, without requiring developers to use specific programming languages or data structures, and without trusting any software except KLINT.

1 Introduction

Network operators are moving from hardware *network functions* to software ones for flexibility, but still deploying them as black boxes. Historically, network functions such as firewalls, NATs, and load balancers used special-purpose hardware for performance at the cost of flexibility, but this trade-off does not always make sense any more thanks to the performance of modern general-purpose hardware. Developers write software network functions using programming languages such as C or Rust and frameworks such as DPDK [15] or BPF [11, 33]. Marketplaces for distribution of software network functions are emerging [18], and most network functions on these marketplaces are proprietary and distributed in binary form.

Developers and operators currently test network functions hoping to catch bugs, but this is not enough especially in light of frequent software updates. For instance, network address translators from Microsoft [36], Linux [37], and Cisco [38] have had headline-causing bugs and vulnerabilities.

```
function Firewall_Check(packet, flow_table)
    if packet.device.is_internal then
        assert packet.is_forwarded
        if packet.flow not in flow_table then
            assert flow_table.was_full
    else
        if packet.is_forwarded then
            assert packet.flow in flow_table
```

Algorithm 1: Specification for a firewall. The firewall must remember flows exiting the network and must only allow packets in an existing flow to enter the network.

Operators need guarantees that the software network functions they deploy conform to specifications, instead of relying on tests. Consider the specification for a firewall in Algorithm 1, which will be our running example. This specification restricts what a firewall can do but does not state exactly how a firewall should be implemented. It also hides details irrelevant to operators such as packet parsers and policies to expire old flows. If an implementation could be *verified* to conform to this specification no matter what, an operator could deploy that implementation with confidence.

Developers need to provide proofs of functional correctness but do not always want to disclose their source code, meaning verification must be done on binaries. This requirement is driven by the way software is distributed to operators, but it also means developers do not need to restrict which programming languages they use nor worry about latent bugs in the tools they use such as compilers. A tool that can verify binaries can be used by operators regardless of whether they have access to source code and regardless of which language and toolchain developers used. Thus, even for open-source code, verifying binaries helps developers by enabling them to use any language and toolchain, even those considered experimental, since they can provide guarantees about the compiled binary to operators.

Developers currently cannot provide proofs (1) of functional correctness, (2) without source code, and (3) without verification expertise. They can use the Linux kernel's automated BPF verifier, but only for low-level properties such as memory safety. They can use Vigor [53] or Gravel [55] to automatically prove functional specifications, but both rely on typing information and thus require source code or an intermediate representation that can be reverse-engineered. Vigor and Gravel also limit developers to specific data structures; while developers could add new ones to the tools, they generally lack the verification expertise to do so. What remains is conventional testing, which can only show that a binary works in specific cases, not provide guarantees.

Verification of data structures is out of scope for this paper. We assume that data structure code in binaries is clearly delineated and that it correctly implements its API and specification. Developers should then be able to write code using any data structure they want. A verification tool for network functions must understand the semantics of data structures through some form of contract, and automatically reason about their contents without requiring proof annotations. Developers who wish to provide maximal guarantees can manually verify their data structures, or use existing verified ones, but this should not be required for network function code to be automatically verifiable.

Our goal is to design a tool to automatically prove that a network function binary conforms to a specification, given data structures assumed to be correct. The inputs to this tool are a binary with explicit calls to data structure operations, contracts for these operations, and a specification such as a formalization of an RFC or IEEE standard. The output is a proof that the binary refines the specification or a counter-example that demonstrates otherwise.

Verifying arbitrary binaries is hard since type and control flow information are critical to verification but hard to obtain without debug symbols. However, network functions are not arbitrary. They commonly confine complex code to well-defined data structures, such as BPF maps [2], and only have a handful of well-defined interactions with their environment, such as transmitting packets and reading system time. These two observations lead to two ideas enabling us to verify network function binaries.

First, we use maps as a "universal" data structure to specify network functions and data structures. We define the abstract state of both network function specifications and data structure contracts in terms of maps from keys to values using a programming language such as Python. We call these maps *ghost maps* by analogy to "ghost variables" that exist only in proofs, not in implementations. Contracts can be written even for data structures that cannot be implemented in terms of maps, by using "for all" quantifiers to describe an operation's effects on the data structure in a declarative manner without describing its implementation. Verification is thus proving that the binary manipulates concrete data structures in a way that conforms to the manipulation of abstract maps in the specification, using contracts to match concrete operations with abstract ones.

Ghost maps enable efficient and automated invariant inference. Inferring invariants ensures automated analysis does not explore impossible program states, which would cause spurious verification failures. The sweeping simplification of maps enables our tool to reason about invariants across any data structures instead of limiting itself to specific ones. It also leads to the tool being simpler, as there is only one kind of data structure to reason about, the map. The tool translates data structure operations into map operations and infers invariants on the resulting maps.

Second, we infer typing and control flow information from environment interactions. Modeling network functions' environment precisely is feasible since it is small and well-defined. Typing information at the boundaries between a binary and its environment is enough, since specifications are defined in terms of environment interactions: maps and fundamental operations such as packet transmission.

We have built KLINT, a prototype of our technique. It takes in a binary without debug symbols, a Python specification of its semantics, and Python contracts for its data structures. The contracts only need to be written once per data structure implementation, as a more precise form of the documentation developers currently write. KLINT always terminates with either a proof or a counter-example, unless the processing of a single packet does not terminate, in which case KLINT fails after a user-defined limit. KLINT can verify an entire software stack using a minimal operating system if needed. KLINT can verify code written in different languages, such as C or Rust, on different frameworks, such as DPDK or BPF, and deployed in different contexts, such as virtual machines, containers, or raw hardware. Developers can use any data structure as long as it has Python contracts, even if the data structure implementation is not verified. The code of the verified network functions is *verification-agnostic*: only the standard programming practice of separating data structures from other code and documenting their specifications is necessary.

We use KLINT to verify network functions we write based on those from Vigor [53] and to verify existing BPF network functions such as Facebook's Katran [47]. KLINT can prove a range of properties, from full functional correctness according to a specification, to memory safety and crash freedom for functions without a high-level specification such as Katran. For instance, we extract a specification from IEEE 802.1D [30] for a bridge we write. We provide a detailed list of properties we verify using KLINT in Appendix A. Using KLINT gives operators guarantees about the correctness of any binary they deploy: either KLINT finds bugs, which can be reported to developers, or it finds no bugs, providing a guarantee the binary conforms to its specification.

In summary, our contributions are (1) a technique to reason about data structures and infer invariants based on the idea of "ghost maps" and (2) KLINT, a prototype that uses map-based reasoning to automatically and efficiently verify network function binaries that use trusted data structures with map-based contracts. We use KLINT to verify the functional correctness of 7 binaries, 6 written in C and 1 in Rust, in minutes. These verified binaries run faster than previous verified ones thanks to relaxing restrictions imposed by previous work. We also verify the memory safety and crash freedom of 5 existing BPF network functions using KLINT, though we have no specifications for them.

The code for KLINT and our network functions is publicly available at https://github.com/dslab-epfl/klint.

2 Design Insights

We formalize what we mean by a "network function" in §2.1, explain how environment interactions are enough to infer typing and control flow information of binaries in §2.2, and describe how we bridge the gap between implementations and specifications through the indirection of maps in §2.3.

2.1 Network functions we target

We restrict ourselves to network functions that use mutable state to process packets and contain two phases: initialization and packet processing. When processing a packet, network functions decide whether to transmit packets in response and what data to transmit based on state and on the packet's contents and metadata, such as its length. We assume that network functions are single-threaded, and that they only execute code when initializing and receiving packets. We do not support timers as triggers to run code; they must instead be checked during packet processing.

To create, read, and update state, network functions acquire and use data structure *capabilities* through an environment, such as the DPDK [15] framework. These capabilities correspond to opaque pointers in C, preventing network functions from directly accessing data structure internals.

Acquiring capabilities is only allowed during initialization, and may fail due to external factors, for instance running out of memory. Using capabilities is allowed at any time, and may only fail through incorrect use, for instance indexing arrays out of bounds. Packet processing may also use other environment calls that cannot fail from external factors, such as obtaining the current time.

Our assumptions about data structures correspond to good programming practices: developers should use data structures in a modular fashion for maintainability, and network functions should not allocate memory while processing packets, to avoid performance issues and out-of-memory errors that could allow for denial-of-service attacks.

We represent our model of a network function in Figure 1. This is only a formalization of what developers already do, not a proposal of a new model.
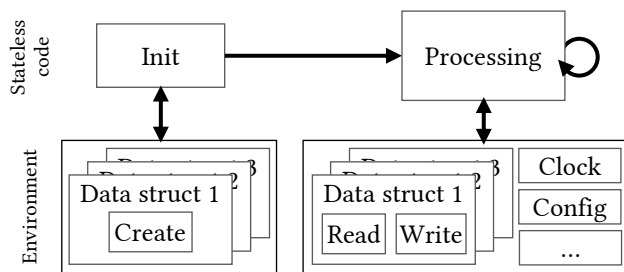

Figure 1. Network function components: initialization allocates state, packet processing reads and writes state and may also use other utilities. Arrows represent control flow.

2.2 Information from environment interactions

The first limitation of previous work that we overcome is the need for source code, thanks to the insight that environment interactions are sufficient to infer all necessary information about network function code, i.e., types and control flow.

As we described in §2.1, the only way for a program to hold state and interact with the external world is through its environment, such as displaying information or receiving a packet from the network. That is, we define the environment to be low-level enough that programs can be modeled as pure functions that compose environment interactions.

A tool can precisely infer all possible behaviors of a program by replacing the program's environment with one that can exhibit any allowed environment behavior, not a specific one. This verification-only environment can be written by hand or inferred from machine-readable documentation.

Writing an exhaustive model of the environment is not feasible for general-purpose software because its environment is large and ill-defined. There is no formal definition of large environments such as operating systems, and documentation often omits implementation details of high-level operations that programs may rely on in practice.

However, a complete environment description is feasible for network functions, because their environments are small and well-defined. Formally defining all environment interactions is manual work but must only be done once by an environment's developers, and the operations are low-level enough that emulating all possible implementation-specific behaviors is feasible. Having a small and well-defined environment also means that its modules can be formally verified manually if its developers choose to do so, enabling verification of the entire software stack.

For instance, our example firewall calls a data structure library to perform a lookup in its flow table and calls a networking framework such as DPDK to forward or drop the packet based on the lookup result. Both the data structures and the networking framework are part of the environment, and both can be precisely replaced by a tool as long as the network function uses dynamic linking.

Environment interactions can be observed on binaries without loss of information, since the tool knows the number of parameters and their types. For instance, when a binary calls DPDK's packet transmission function, the tool uses the current architecture's calling convention, which is known, and the data types in DPDK's headers, which are publicly available, to extract the arguments to the transmission function from the current machine state, such as the packet buffer and metadata. Thus, source code is not necessary for automated software network function verification.

The control flow information necessary for verification can be similarly extracted by observing which branches are taken or not between environment interactions, which produces an "unrolled" version of the true control flow graph.

2.3 Using maps to bridge the gap

Verifying the correctness of a network function that satisfies the definition in §2.1 consists of showing that the semantics of the code, obtained from environment interactions as described in §2.2, match the semantics of the specification. However, there is a gap between the variety of data structures that exist in the code and the restricted set of abstractions that any given tool can reason about, as we illustrate in Figure 2.

For instance, our firewall may use a "least recently used" data structure to keep track of which packet flows should be expired due to a lack of activity, with operations such as "add a new item" and "remove and return the oldest item". The firewall may also combine the use of this data structure with other structures, such as a map tracking per-flow statistics or a configurable set of ports that are open to the external world at all times.

Previous work proposed two ways to bridge the gap between implementation and specification: Vigor [53] requires the use of specific data structures in both implementation and specification, while Gravel [55] requires the use of specific data structures that it knows how to model in terms of high-level operations that can be used in specifications.

Vigor imposes two constraints to verify our firewall. First, the firewall must be written using data structures that Vigor knows about, either by modifying the firewall's code to only use Vigor data structures or by modifying Vigor itself to handle new data structures, including proof annotations for invariants that can hold across these data structures such as "all flows in the LRU are also in the map". Second, the firewall specification must be written in terms of the same data structures used in the implementation. These two constraints limit both developers and operators. Developers must restrict themselves to specific data structures or learn verification techniques to add new ones, and operators must learn the semantics of the data structures used in an implementation to understand its specification.

Gravel removes the second limitation: it translates data structure operations to a small set of high-level operations for use in specifications, thus operators do not need to learn all data structure semantics.
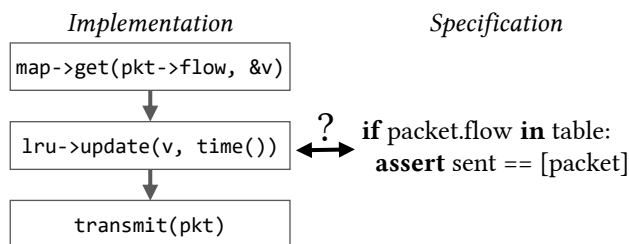

Implementation   *Specification*

```
map->get(pkt->flow, &v)
        ↓
lru->update(v, time())    ↕?    if packet.flow in table:
        ↓                            assert sent == [packet]
    transmit(pkt)
```

Figure 2. There is a gap between the abstractions used in the implementation and the specification.

However, the restriction on developers remains. Developers must either modify the firewall's code to only use existing Gravel data structures or modify Gravel itself to handle new data structures.

Fundamentally, tools that handle a specific set of data structures do not scale. Adding support for a new data structure requires not only encoding the data structure's operations, but also its interactions with other data structures, such as invariants that can exist across structures. Even if a tool is limited to invariants across two data structures, adding the Nth structure requires adding N kinds of invariants, one for each data structure including the new one.

We introduce a level of conceptual indirection: we express the semantics of both data structures and specifications in terms of one data structure, the *map*. Operations on data structures such as arrays, hash tables, longest-prefix-match tables, and port allocators can be defined in terms of map operations, regardless of how they are implemented. Verification tools can use contracts to translate any data structure operation into map operations, which become the only kind of operation the tool needs to reason about for invariant inference and verification.

We refer to such maps as *ghost maps*, by analogy to "ghost variables" which are variables only used in proofs. We present an example of contracts for a "least recently used" data structure in Listing 1. Using this contract, a verification tool can translate the LRU semantics into maps, and thus does not need special knowledge of what an LRU data structure is, only knowledge of maps. Importantly, contracts can be declarative, not just imperative. Ghost maps can define even data structures that cannot be implemented using maps thanks to the "for all" quantifier. LRU_expire's contract only describes *what* it does, not *how*: the returned value is the oldest, but the contract does not need to explain how this value is found. We explain how a verification tool can efficiently reason with maps in §3.

This sweeping simplification also makes invariant inference easier since there is only one kind of data structure, as we show in §4.

```
1  # struct LRU;
2    LRU = namedtuple('LRU', ['items'])
3  # void LRU_add(struct LRU* lru,
4  #              void* item, int age);
5    assert item not in lru.items
6    lru.items[item] = age
7  # void* LRU_expire(struct LRU* lru);
8    age = lru.items[result]
9    assume(lru.items.forall(lambda k, v: v <= age))
10   lru.items.remove(result)
```

Listing 1. Contracts defined using maps, in Python, for a "least recently used" data structure, in C. result is the return value from LRU_expire.

3 Ghost Maps

To verify network functions automatically, we use *symbolic execution* to analyze all their paths, which we summarize in §3.1. Since data structure implementations have too many paths to enumerate, we abstract them using *contracts* instead, to obtain paths such as "key found" and "key not found" instead of one path per cell in which a key might be.

We introduce ghost maps as a "vocabulary" in contracts to describe the semantics of data structures and network functions to both verification tools and human readers. Ghost maps only exist within contracts, not implementations.

We describe our goals for ghost maps in §3.2, our proposed representation of ghost maps in §3.3, and our proposed translation of their operations into logical formulas in §3.4.

3.1 Symbolic execution background

A *symbolic execution engine* executes code with *symbols* as inputs instead of concrete values. Whenever it encounters a branch on a symbolic condition, it explores both alternatives, remembering the choices it made in a *path constraint*. This leads to a set of *paths*, which are sequences of choices that each represent one possible program execution. For instance, instead of executing an "absolute value" operation on -5 and obtaining 5, a symbolic execution engine will execute it on α and obtain two paths: one with path constraint $\alpha \geq 0$ and result α, and one with path constraint $\alpha < 0$ and result $-\alpha$.

Some code has too many paths to explore in reasonable time. For instance, consider looking for a value in an array. The value could be in the first position, or the second, or the third, and so on, until the array length. The number of paths is limited by the array length, which could be, e.g., 2^{32}. If the code which looked up the value then looks up another value, each path resulting from the first lookup will lead to new paths for the second lookup, squaring the total number of paths. This problem is known as *path explosion*. While the number of paths can sometimes be reduced by merging related paths at the expense of producing more complex constraints [29], the paths to be merged must still be partially explored, which does not solve path explosion.

If the number of paths in a program is "reasonably" small, a symbolic execution engine can exhaustively enumerate them and verify the program by verifying that each path satisfies the program's specification.
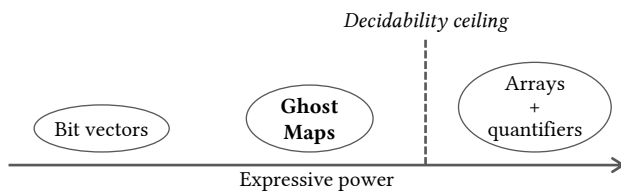

Figure 3. Ghost maps are more expressive than bitvectors while remaining decidable, unlike more powerful theories such as arrays with universal quantifiers.

$$length(M) \rightarrow Int$$
$$get(M, K) \rightarrow V \mid None$$
$$set(M, K, V) \rightarrow M'$$
$$remove(M, K) \rightarrow M'$$
$$forall\big(M, \lambda(k, v) \rightarrow Bool\big) \rightarrow Bool$$

Listing 1. Ghost map operations. M and M' are maps; K, V are keys and values. *Int* denotes bitvector-based integers, and *Bool* Booleans. *None* is the lack of value.

3.2 Expressivity, decidability, and completeness

We propose the ghost map abstraction in Listing 1 which is expressive enough to define data structures and network functions, while still enabling a tool to reason in a decidable, sound, and "complete enough" manner. We describe each of these properties next.

Ghost maps are *expressive* enough to abstract the data structures we care about. Simpler abstractions require too much detail in contracts to be practical for either humans or tools. For instance, representing a hash table as a sequence of 0s and 1s is possible but impractical. We are concerned with data structures used in network functions, such as hash tables and port allocators, thus we limit our vocabulary to what they need, not to all possible code.

However, expressiveness is at odds with *decidability*. Symbolic execution engines use a solver to tell whether a logical formula is *satisfiable*, i.e., whether there exists an assignment of variables such that the formula holds. For instance, if "the firewall's variables, given the firewall's constraints, violate its specification" is satisfiable, then the assignment of variables is a counter-example to the firewall's correctness. Logical formulas are written using *theories*, which are the "vocabulary" of solvers. Some theories are *decidable*, meaning that a correct solver will always return a correct answer. Some are not, meaning that the answer may be "unknown" instead of yes or no. Even with decidable theories, "unknown" may be returned if the solver is given a timeout and cannot find an answer in time.

Verification tools must be *sound* and as *complete* as possible. A tool is sound if it verifies *only* correct programs. A tool is complete if it verifies *all* correct programs. Due to the Halting Problem [49], verification of general-purpose programming languages must be incomplete, thus the goal is to verify "interesting" correct programs, i.e., those that humans actually write, even if some contrived theoretical examples cannot be verified.

Ghost maps are an intermediate step between quantifier-free bit vectors, which are decidable due to their finite size but not expressive enough, and arrays with universal quantifiers, which are more than expressive enough but undecidable, as we illustrate in Figure 3.

3.3 Representing ghost maps

To remain as decidable as the theory of bit vectors while offering more expressivity, we present a translation of ghost maps to bit vectors in the context of symbolic execution. Notably, ghost maps' "for all" quantifier, which enables non-imperative contracts for data structures that cannot be implemented with maps, can be translated *without* using universal quantifiers. Ghost maps can be more expressive despite being translated to bit vectors because the symbolic execution engine internally uses data structures, such as lists, to build the logical formulas it sends to the solver.

We expect the code to manipulate maps of large size, but to only interact with a small number of items in any given map. This is true of network functions, which by nature only perform a small number of operations for each packet to remain within their performance budget.

Tracking *known* and *unknown* items separately is our key insight to handle maps of arbitrary size. That is, none of the map operations require "forking" the current path, and the engine handles map operations in a time linear in the number of known items, not total items.

Thus, instead of keeping track of every item in every map, the engine only needs to track the specific items that are explicitly used in one iteration of the network function's packet-processing loop. Other items are "summarized" into a single pseudo-item that tracks their constraints, such as "all unknown values are non-zero".

This scheme naturally enables ghost map lengths to be symbolic, since their actual size does not matter as much as the number of known items. A verification tool can thus represent maps whose length is determined by configuration parameters using a symbolic configuration, instead of verifying only one specific configuration as is for instance done in Vigor [53]. The tool can thus catch all bugs that only occur in specific configurations, such as the maximum capacity of the firewall's flow table being zero, without requiring developers to think of which configurations to try.

Counter-intuitively, the engine must mutate its internal representation of maps during read-only operations on maps. Known items must include those that have been retrieved from the map, even if they have not explicitly been set before. For instance, consider the following:

$$get(M, K_1) \rightarrow V_1$$
$$get(M, K_2) \rightarrow V_2$$

If $K_1 = K_2$, then $V_1 = V_2$ by definition. But the engine must remember the first *get* in some way in order to guarantee the implication, and it cannot store high-level map operations in the path constraint, thus it must modify its internal representation of M. From an outside perspective, M has not changed, but internally the engine must remember this *get*.

We informally describe each operation first, then add additional details to handle subtleties, then provide a formal algorithm for the core *get* operation.

The engine tracks each map's length explicitly.

Known items are triples: (key, value, presence bit). If the presence bit is *false*, the value is ignored and the key is considered absent from the map. *None* only exists conceptually; the theory of bit vectors cannot represent it.

Known items may be redundant, but their values and presence bits must match if their keys match. This is because their keys may be symbolic, thus the engine cannot know for sure whether two items have equal keys.

Unknown items are represented by an *invariant* that they all satisfy. The unknown items invariant only concerns unknown items. Known items need not satisfy it. For instance, a map may have as invariant "all unknown values are non-zero" and two known items, $(K_1, 0, true)$, $(K_2, 1, \alpha)$. The first known item is definitely present, whereas the second may or may not be present depending on the value of α.

The unknown items invariant is represented as a formula on a special *unknown item*, unique to each map. For instance, the non-zero invariant example is represented as $UP_M \Rightarrow UV_M \neq 0$, where UP_M and UV_M are the presence bit and value of M's unknown item. Including the presence bit in the invariant allows it to be constrained for cases such as arrays. For instance, a zero-based array of length L described using a ghost map A would have $UP_A = (0 \leq UK_A < L)$ as part of its unknown items invariant, indicating that keys are in A if and only if they are between 0 and L, matching the semantics of array indexing in languages such as C.

3.4 Translating ghost map operations

We use $ITE(c, t, f)$ to denote "if c then t else f", and *fresh* to mean a symbol that was not used before, i.e., that is not constrained in any way. "Applying" the unknown invariant of a map to an item means substituting the map's unknown item for the item.

$length(M)$: Return the map's length, which the engine tracks explicitly.

$get(M, K)$: Create a fresh tuple (V, P). Add (K, V, P) to the map's known items. Add constraints to the current path to encode that, within the map, (1) if K matches a known item then so do V and P, (2) if K does not match any known item then the unknown items invariant applies on (K, V, P), and (3) the number of unique known items cannot exceed its length. Return (V, P), which encodes "if P then V else *None*".

$set(M, K, V)$: Let $(\_, P) = get(M, K)$. Return a new map whose length is $length(M) + ITE(P, 0, 1)$, whose known items are $(K', ITE(K = K', V, V'), P' \vee (K = K'))$ for each known item (K', V', P') in M, plus $(K, V, true)$, and whose unknown items invariant is the same. That is, (1) the length only grows if K was not in M, and (2) known items must match if they are redundant.

$remove(M, K)$: The opposite of *set*, i.e., the length change is $ITE(P, -1, 0)$, the new known item is $(K, V, false)$, where V is any arbitrary value, and the known items are changed to $(K', ITE(K = K', V, V'), P' \wedge (K \neq K'))$.

$forall(M, F)$: The result is *true* iff $P \Rightarrow F(K, V)$ for each known item (K, V, P) in the map and $UP_M \Rightarrow F(UK_M, UV_M)$ for the unknown item of the map. That is, the result indicates whether the predicate holds on known items and on unknown items. Add to the map's invariant that if the result of this operation is *true*, then the predicate holds on unknown items if their presence bit is *true*, ensuring that even if the result is not yet known, it will be consistently applied in the future. If the result is false, future items are not constrained, which is sound but not complete.

***Layers* are necessary to handle dependencies** that arise when code uses multiple versions of a map at once. Consider:

$$set(M, K, V) \rightarrow M' \quad \begin{array}{l} get(M, K') \rightarrow V' \\ get(M', K') \rightarrow V'' \end{array}$$

If $K \neq K'$, then V' and V'' must be the same, since only the value associated with K is affected by the *set*. However, the representation described earlier cannot guarantee this, because the known items of M and M' are independent.

To handle this issue, *set* and *remove* return layers instead of entirely new maps, as the example in Figure 4 illustrates. That is, they return a map which includes the newly added or removed item among its known items, but which links to the previous map for the other known items, transforming them when necessary, instead of evaluating the new known items at creation time. Items and invariants created by *get* and *forall* are always added to the bottom-most layer. Map layers also share the unknown items invariant, and the associated "unknown item", of their base map.

Thus, in our example above, the known items "seen" by the second *get* operation include the result of the first one, and thus the second *get* will return a result that lets a tool prove $K \neq K' \Rightarrow V' = V''$.

Invariant recursion must be explicitly handled to avoid infinite recursion when two maps' invariants refer to each other. For instance, a bi-directional map may be represented as two maps whose contents are inverses: for each key-value pair (K, V) in map M_1, there is a pair (V, K) in map M_2, and vice versa. The invariants are:

$$forall(M_1, \lambda(k, v). \; get(M_2, v) = k)$$
$$forall(M_2, \lambda(k, v). \; get(M_1, v) = k)$$

Consider what would happen using the representation described earlier when the engine handles $get(M_1, K)$ for some K. As part of adding the invariant on the newly-known item of M_1 to the path constraint, the engine will call $get(M_2, V)$ with the fresh V from the original *get*. As part of adding the invariant on the newly-known item of M_2 to the path constraint, the engine will call $get(M_1, V')$ with the fresh V' from the second *get*. The engine then calls *get* on M_2, and so on, leading to infinite recursion.

Solving this issue requires the engine to recognize that in the third *get* call, the V' argument is equal to K, which it knows from the first *get* call. The result should thus be the existing V, not some fresh V''.
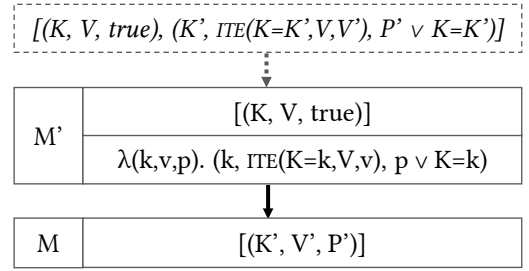

Figure 4. Example of a *set* layer M' on top of a map M, and the resulting known items of M'.

The engine thus tracks a *condition* and a *value hint* during ghost map operations, which are set when handling invariants and used to stop recursion when handling *get*.

When handling a map invariant of the form $get(...) = ...$, the engine adds to the condition the presence bit given as argument to the invariant and sets the value hint to the value expected by the invariant. These changes are reverted when the engine is finished handling the invariant.

$get(M, K)$ needs two changes at the start: First, if K cannot be different from an existing item's key assuming the condition holds, return that item's value and presence bit. Second, after creating the fresh V, if there is a condition, add "the condition implies $V = value\_hint$" to the path constraint.

Applying this logic to our example solves the issue. When handling $get(M_2, V)$, the value hint is K and the condition is P, both from the newly-known item of M_1. When M_2's invariant calls $get(M_1, V')$, the path constraint contains $P \Rightarrow V' = K$, thus the *get* on M_1 will start by checking whether its known item (K, V, P)'s key is equal to V' assuming P, which it is, and return (V, P), ending the recursion.

This strategy avoids recursion in common cases such as our example of maps with reciprocal keys and values, but it is not complete, as the engine may recurse infinitely. For instance, "M_1 has $min(K - 1, 0)$ for all K in M_2, and M_2 has $min(K - 1, 0)$ for all K in M_1" could hold, but will lead to infinite recursion in the engine given our implementation.

We present the final *get* algorithm, which is the core of our ghost map technique, in Appendix B. Our technique is decidable and expressive but not complete, unlike prior work that focused on completeness at the expense of expressiveness [3, 13]. Our technique enables a symbolic execution engine to translate ghost map operations into formulas on quantifier-free bit vectors. This enables the engine to bypass the path explosion caused by data structure code by executing the code's contract instead.

The universal "forall" quantifier on ghost maps allows contracts even for operations that cannot be implemented using maps by describing *what* they do instead of *how*. However, we believe some data structures are not a good fit for ghost maps, specifically ordered ones. While queues and stacks can be viewed as maps from indexes to elements, the resulting contracts are unlikely to be conducive to invariant inference.

4 Invariant inference

Handling data structures by translating them to ghost maps is not sufficient for automated verification. Developers use data structures in well-defined patterns, named *invariants*, but these patterns are not always explicit in the code.

Verification tools must *infer* such invariants to avoid failures. When executing a contract instead of an implementation, the tool must have enough information to prove the contract's precondition. Previous tools bypassed this problem by special-casing data structures and their invariants.

The ghost maps representation we proposed in §3 enables tools to infer invariants without special-casing templates.

Consider these motivating examples in pseudo-code:

```
if packet.flow in table:
    statistics.increment(packet.flow)
```

If `increment`'s contract requires the item to be in `statistics`, symbolic execution will fail if it cannot prove this fact.

```
device = destinations.get(packet.flow)
transmit_packet(packet, device)
```

If `transmit_packet`'s contract requires the device to exist, symbolic execution will fail if it cannot prove this fact.

```
if not items.full:
    items.add(x)
    metadata.add(x, y)
```

If `add`'s contract requires free space, symbolic execution will fail if it cannot prove `not` `metadata.full` in the second `add` call.

All three examples could be bugs depending on how the data structures they deal with are updated. If the first example is in code that always adds `packet.flow` to both `statistics` and `table`, the code is valid. If the second example always puts a device known to be valid in `devices`, such as the incoming device of a packet, the code is valid. If the third example is the only occurrence of `add` in the code and both structures have the same length, the code is valid.

To infer invariants, our algorithm starts from the strongest possible ones then iterates by relaxing them as needed until it finds a fixed point, as we illustrate in Algorithm 2. The starting point is the program states resulting from symbolically executing the network function's initialization code. At this point, the invariants are the strongest possible ones: "all maps will always be exactly as they are after initialization".

---

function FindFixedPoint(*code*)
  invs = *InitialInvariants*(*code*)
  states = *InitialStates*(*code*)
  do
    invs = *Relax*(*invs*, *states*)
    states = *SymbolicallyExecute*(*states*, *invs*)
  while *invs* do not hold on *states*
  return *invs*

Algorithm 2: Core of the invariant inference algorithm.

The initial invariants are unlikely to hold on the packet processing code, unless the code does nothing. After symbolically executing the packet processing code for one iteration of the network function's infinite packet-processing loop, the engine *relaxes* the invariants to match the program states that result from the iteration. For instance, the initial invariant "the firewall's `flow_table` is always empty" could be relaxed into "the `flow_table` may have items, and its length is always the same as the `statistics`". The engine then symbolically executes a packet-processing iteration again using these new invariants, which may lead the engine to explore new paths in the code such as a path in which the packet's flow is found in the `flow_table`, previously infeasible as the table was assumed to be empty. The engine then relaxes the invariants again, and executes an iteration with these new invariants, until the invariants no longer need relaxing after an iteration. By definition, the final relaxation yields invariants that hold on the initial state as well, thus the result is a correct set of invariants. The algorithm is guaranteed to converge since the set of invariants can only shrink. It may converge towards the empty set of invariants because the code has no invariants or because the engine could not infer any. The goal is not to find some "ideal" set of perfect invariants, only enough invariants to be able to symbolically execute and verify the code. If some properties happen to hold but are not necessary, inference need not find them.

The key to inferring useful invariants is to use ghost maps' *known* items to form invariant candidates, and then check whether these candidates hold using the *unknown* items. Thus, instead of using low-level invariant templates such as "the value is nonzero" as in Houdini [20], the tool can find invariants based on the constraints that hold on known items. For instance, the tool does not need a definition of a "device" to solve our second motivating example.

Besides finding invariants among map items and lengths, tools can also find invariants *across* maps. Such invariants are of the form "if M_1 associates key K with value V, then M_2 has some key and possibly value related to K, V". For instance, in the first motivating example, finding the invariant "all keys of `table` are also keys in `statistics`" enables the tool to prove that the code uses `increment` correctly. Inferring such invariants requires finding functions F_K, F_V, F_P for two maps M_1, M_2 such that $((get(M_1, K) = V) \land F_P(K, V)) \implies (get(M_2, F_K(K, V)) = F_V(K, V))$, or alternatively find only F_K, F_P and merely infer that M_2 contains $F_K(K, V)$ without inferring the associated value. Candidates for F_* are found using the known items and confirmed using the unknown items invariant.

In summary, the representation we propose for ghost maps also enables an elegant invariant inference algorithm based on finding candidate invariants and relaxing them as necessary. Representing known items explicitly lets a verification tool find useful candidates for invariants, instead of limiting itself to special-cased low-level templates.

5   Implementation

We implement our technique in KLINT, a tool which uses the angr [48] symbolic execution engine and the Z3 [12] solver.

KLINT takes a network function binary and a Python specification as inputs and proves that the binary satisfies the specification or produces a counter-example. KLINT can also be used without a specification to check that no path in the code crashes or accesses memory out of bounds.

KLINT first identifies all environment interactions in the binary, which correspond to data structure operations or networking operations, using the symbols that the binary must export since it dynamically links the environment code. Then, KLINT maps these operations to their contracts, provided by the developers of the data structures and of the networking environment. Data structure contracts are written in terms of ghost maps, as defined in §3, while the handful of fundamental networking operations such as transmitting a packet have manually written contracts for KLINT. When the binary calls its environment, KLINT symbolically executes the corresponding contract instead of the implementation, inferring types and control flow information as described in §2.2: KLINT knows the types of parameters to environment interactions, and extracts control flow in the form of path constraints. This enables KLINT to understand the semantics of the binary under verification in terms of its environment.

KLINT symbolically executes the network function's initialization code, as defined in §2.1, then infers invariants that hold in all packet-processing paths by symbolically executing the packet-processing code until it finds a fixed point, as described in §4. KLINT then checks whether all packet-processing paths satisfy the specification, which indicates whether the network function binary as a whole does.

Specifications are Python programs that use ghost maps, which KLINT interprets using peer symbolic execution [6] on each state resulting from symbolic execution of the binary. The binary may abstractly manipulate more than one ghost map, since it may use multiple data structures and individual data structures may be modeled by contracts as multiple maps. Thus, when the specification refers to a map, KLINT must infer which of the maps it is, using heuristics to try likely candidates first, and only fail verification if the specification is violated for all possibilities. For instance, a firewall can track outgoing flows and maintain statistics per IP address. KLINT must infer that the flow table in Algorithm 1 is the abstract form of the former data structure, not the latter.

Developers must comply with good programming practices such as state separation if they want verification to succeed, and KLINT enforces this during verification. If developers do not separate data structure code from the rest of the network function, KLINT will encounter too many paths and fail once it has executed a configurable instruction threshold. Developers already follow even stricter practices to write BPF programs due to the strict Linux verifier [10].

KLINT models heap memory with ghost maps, treating memory as just another kind of data structure and including it in invariant inference. Developers use a standard calloc-like interface to allocate memory. During symbolic execution, memory allocations return *symbolic pointers*, ensuring that KLINT will explore all paths even if some paths depend on the value of pointers. This is not the case in a tool such as Vigor [53], which uses concrete pointers.

To ensure that developers cannot "hide" memory from KLINT, all memory outside of the stack and the maps-modeled heap is read-only after initialization. This is not a limitation on sensible developers, who allocate on the heap through the environment and cleanly separate mutable state.

Our memory model is similar to Memsight [9] and KLEE's segmented memory [26], but it requires almost no effort to implement thanks to the flexibility of ghost maps.

KLINT can do *full-stack* verification, verifying the entire software stack, including the network driver and most of a minimal operating system.

To verify the network driver, KLINT matches hardware accesses to the *actions* they correspond to, using a hardware description of the network card written in a domain-specific language and based on publicly available data. KLINT intercepts reads and writes to network card registers, which usually go through port- or memory-mapped I/O. When the network driver writes a value to a register, KLINT reverse-engineers what the driver might be doing by finding all actions that could match the write, and checking which ones are feasible in the current hardware state. For instance, if the driver sets the "enable promiscuous mode" flag in the network card, KLINT looks up the corresponding action and checks its precondition, which states packet reception must be disabled. If the precondition does not hold, or if no action matches the write performed by hardware, KLINT aborts verification. Actions can also have postconditions describing what happens to the hardware as the result of an action, such as a self-clearing bit in a register.

The only environment operations for which KLINT uses contracts instead of implementations during full-stack verification are the data structures and the memory allocator due to their complexity. We verified that they obey their contracts using machine-checked proofs.

Full-stack binary verification does impose one constraint: while the network function and its network driver can be compiled together, the environment has to be compiled separately and then linked together without link-time optimizations, ensuring the symbols corresponding to environment operations exist and can be given as an input to KLINT.

Our trusted computing base is made up of KLINT itself, including angr and Z3; the bootloader; the hardware; and the VeriFast [25] theorem prover we use to verify the memory allocator and data structures. Since VeriFast verifies source code, we trust the C compiler for the data structures and memory allocator, but this is not a fundamental limitation.

6 Evaluation

We evaluate KLINT using the binaries, without debug symbols, of 6 network functions: an Ethernet bridge with a spanning tree protocol, a firewall, an implementation of Google's Maglev [16], a network address translator, a traffic policer, and an IPv4 router with longest prefix matching. The first five are based on publicly available code from the Vigor [53] project, which verified source code.

We show that KLINT can quickly verify binaries in §6.1, that it reduces the trusted base and enables developers to write faster code in §6.2, and that it is applicable to real-world code including BPF in §6.3.

6.1 Verifying network functions

We summarize the time it takes KLINT to verify our network functions in Table 1, split into the time to symbolically execute the code, infer invariants, and verify the resulting paths. KLINT runs multiple iterations of symbolic execution and invariant inference to find a fixed point, thus we report the total time spent in each category.

We measure all times on an Intel i7-7700HQ CPU running at 3.60GHz. KLINT is single-threaded, though invariant inference is embarrassingly parallelizable. We prototyped parallelization but ran into a complex bug between angr and Z3 due to garbage collection [1], and since KLINT is fast enough we did not investigate further.

Overall, even the most complex of our network functions, the bridge, takes about 2 minutes to verify, which we believe is reasonable. We could further reduce verification time by reducing the redundancy in some invariants, and by using less flexible tools than angr and Z3, since we do not use their full power. The router uses a single data structure, the longest-prefix-match table, thus it only has one invariant.

We caution against over-interpreting the exact results, as most of the time is spent by the Z3 solver, and we noticed that the time Z3 takes to solve queries can vary significantly with small changes in queries, due to the heuristic-based nature of solving. Thus, total verification time can vary by dozens of seconds with minor changes in KLINT or Z3.

	Time (seconds)				#invs
	Sym. ex.	Inv. inf.	Verif.	Total	
Bridge	82	18	19	**119**	32
Firewall	26	7	10	**43**	20
Maglev	36	11	10	**57**	25
NAT	43	7	8	**58**	20
Policer	53	10	6	**69**	25
Router	0	0	1	**1**	1

Table 1. Our network functions, the time KLINT takes to verify them, and the number of invariants it finds.

```python
1  def spec(pkt, config, sent_pkt):
2    if pkt.ipv4 is None or pkt.tcpudp is None:
3      assert sent_pkt is None
4      return
5    if pkt.device == config["external device"]:
6      flow = {
7        'src_ip': pkt.ipv4.dst,
8        'dst_ip': pkt.ipv4.src,
9        'src_port': pkt.tcpudp.dst,
10       'dst_port': pkt.tcpudp.src,
11       'protocol': pkt.ipv4.protocol
12     }
13     # there must exist a map that tracks flows,
14     # regardless of what it maps them to
15     table = Map(typeof(flow), ...)
16     if sent_pkt is not None:
17       assert flow in table.old
18       assert sent_pkt.data == pkt.data
19       assert sent_pkt.device == 1 - pkt.device
```

Listing 2. Partial specification of a firewall with 2 devices. This specification can be given to KLINT unmodified.

We show the KLINT equivalent of Algorithm 1, our running example of a specification, in Listing 2. The full specification is too long to show here, but KLINT can also verify this partial specification.

The specification abstracts away implementation details such as the type of values in the firewall's flow table, as seen in line 15. This uses Python's "ellipsis" literal, meant for domain-specific languages.

This specification is actual Python code run in the standard Python interpreter by KLINT, thus developers can write specifications using their existing programming knowledge. KLINT currently requires the order of fields in specification structures such as the flow declared in line 6 to match the order in the corresponding implementation structure. Having KLINT try all possible orders would not finish in reasonable time, but there may be better strategies.

We found and fixed two implementation bugs using specifications we wrote based on standards. First, section 7.8 of the IEEE 802.1D standard [30] forbids adding Ethernet group addresses to the filtering table, which we originally forgot to implement. Second, our NAT originally misused its port allocator if configured to use only a sub-range of ports. This was already present in the Vigor NAT, because Vigor requires a concrete size for data structures during verification, and its authors only verified it for the full port range.

Verifying the entire stack requires an additional 15 minutes per network function, most of which is spent inferring the network driver's actions because the driver performs thousands of writes to registers and each of these writes requires work from KLINT. This could be improved by recognizing simple loops, since most initialization operations are performed in loops over categories of registers.

6.2 Faster verified network functions

KLINT enables developers to write faster verified network functions in two ways: developers can quickly prototype changes to data structures without having to verify them first, and they can use the abstractions they want instead of verification-specific ones. In practice, we expect developers to either use existing verified data structures or write contracts for existing "trusted" data structures such as BPF maps. KLINT enables this workflow by only requiring contracts and not proofs for these trusted data structures, whereas previous tools required an all-or-nothing approach.

First, ghost maps and invariant inference enable quick prototyping of new data structures. For instance, we rewrote our port allocator to have different performance characteristics with slightly different semantics, including stricter preconditions as we believed our network functions did not need the generality of the existing ones. Using an approach such as Vigor or Gravel, even when prototyping, we would have needed to add a model of the new port allocator to the verification tool, including annotations for invariants, to check whether our network functions would still be correct when using it. With KLINT, we wrote new contracts, automatically verified that our network functions already satisfied the new stricter preconditions, then manually verified the implementation once we finished prototyping.

Second, verifying binaries enables simpler and faster code by removing abstractions over low-level hardware features that existed for the sake of verification. For instance, using a tool such as Vigor, obtaining the time from the environment requires calling a C function that the tool replaces with a model at verification time. Unless the compiler can inline this function, the developer will pay the performance cost at run time. With KLINT, the code can use inline assembly to read the CPU's time stamp counter, which KLINT handles in the same way as other assembly instructions. Furthermore, in the Vigor model, the C function must be verified separately using a different tool, a problem KLINT does not have.

	Vigor			KLINT		
	Tput	**Latency** (*us*)		**Tput**	**Latency** (*us*)	
	(*Gb/s*)	50%	99%	(*Gb/s*)	50%	99%
Bridge	5.54	3.98	4.25	10*	3.84	4.26
Firewall	7.77	3.92	4.26	10*	3.84	4.25
Maglev	6.34	3.96	4.28	10*	3.90	4.27
NAT	3.47	3.97	4.32	10*	3.87	4.27
Policer	9.12	3.87	4.25	10*	3.83	4.24

Table 2. Maximal single-link throughput without loss, and latency with 1 Gb/s background load of the original Vigor network functions and our versions.   * = link saturated
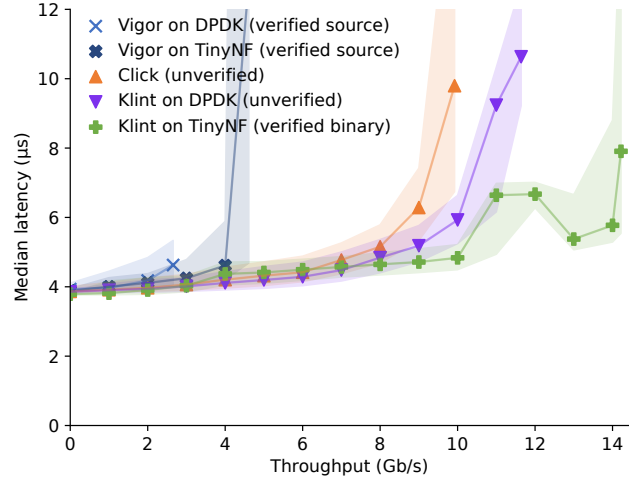

Figure 5. Throughput without loss vs. median latency of different bridges. Shaded areas delimit 5th and 95th percentiles.

We benchmark our network functions using the Vigor ones as baselines, with the TinyNF [45] driver instead of the original DPDK subset since it is faster and is the base for our own network driver. We use the same setup as Vigor to make the comparison useful: two machines as in RFC 2544 [4], one running a network function and one running the MoonGen packet generator [17]. Both machines have Intel Xeon E5-2667 v2 CPUs at 3.30GHz, Intel 82599ES NICs, and run Ubuntu 18.04. These network cards are the ones we modeled for KLINT's full-stack verification. We measure throughput with minimally-sized packets, filling network functions' flow tables to 90% of their capacity.

We first run the same benchmark used in the Vigor paper: find the maximum throughput the network functions can sustain without dropping packets using one 10 Gb/s link and measure their latency when fed 1 Gb/s of background load. All our network functions can handle 10 Gb/s whereas the Vigor ones cannot, as we show in Table 2.

Since the Ethernet link is a bottleneck in the Vigor benchmarks, we use two links, for a theoretical maximal throughput of 20 Gb/s. To obtain more details about performance, we measure latency at increments of 1 Gb/s until the network function drops packets. We focus on the bridge for lack of space. We included the original Vigor bridge running on its verified DPDK subset, the Vigor bridge running on the verified TinyNF driver, the Click [28] bridge originally used as a baseline by Vigor, our bridge running on our verified driver, and our bridge running on DPDK, which is not verified. We do not know of any "standard" network functions we could use as a baseline beyond Click.

Despite our bridge having extra features compared to the Vigor one, such as support for a spanning tree protocol, it can reach more throughput before dropping packets than any of the other bridges, including the bridge from the widely used Click toolchain, as we show in Figure 5.

	Language	Data structures	Extra inputs	Pointer arith.	Unbounded loops	Precise
Gravel [55]	LLVM	C++ STL	Intermediate specs	Limited	No	Yes
Prevail [21]	BPF	BPF maps	No need	If safe	Yes	No
Vigor [53]	LLVM	Custom "libVig"	NF-specific models	No	No	Yes
KLINT	x86_64	Any, w/ contracts	No need	If safe	No	Yes

Table 3. Comparison of this work with previous network function verification efforts.

6.3 Applicability

KLINT is applicable to real-world network functions: it does not require additional inputs from developers, it enables operators to verify the entire stack of network functions, its network function model is a superset of the widely used BPF, and it enables developers to use any programming language including those not considered mature enough to be trusted.

As we show in Table 3, KLINT operates on binaries, can handle any data structure for which map-based contracts are provided, does not require intermediate inputs, does not limit developers' use of pointer arithmetic beyond memory safety, and precisely tracks the contents of data structures and packets enabling functional correctness proofs.

Previous work falls into two categories. Vigor and Gravel prove functional correctness but require a typed intermediate language, extra inputs, and specific data structures. Prevail [21] handles BPF bytecode and maps, which BPF developers must use anyway, and can even handle unbounded loops, but can only prove memory safety and crash freedom. Prevail can be viewed as a superset of the Linux BPF verifier. KLINT provides the best of both worlds: it requires neither a typed intermediate representation nor extra inputs, and it does not limit data structures, while also enabling proofs of functional correctness. As an example of unnecessary inputs, we were able to remove around 3000 lines of proofs for invariants when porting Vigor network functions to KLINT.

Operators can verify the entire software stack, and thus do not need to trust software such as network drivers, using KLINT's full-stack verification. Developers can thus tune the drivers for performance by removing features they do not need, or even rewrite their own driver to suit it to a specific usage pattern. PacketMill [19] showed that such transformations can be done with developer hints to increase network function performance.

KLINT can be used to verify network functions in containers, i.e., statically linked with a Linux implementation of the environment abstraction and running within Docker [34]. Containers are a convenient way to deploy and manage programs and have been proposed as a deployment model for network functions [50]. KLINT can verify such network functions in the same way it verifies full-stack ones, although the trusted code base is larger since operators need to trust the container runtime as well as the Linux environment implementation running inside the container.

Our network function model is a superset of BPF. The Linux kernel verifies that BPF programs are memory safe and have no unbounded loops. KLINT verifies functional correctness, though it also requires a lack of unbounded loops. BPF requires developers to use a fixed set of data structures, mostly maps. KLINT enables developers to use any data structure that has contracts based on ghost maps.

To show that KLINT's model is at least as expressive as BPF's, we use five existing BPF programs: Facebook's Katran [47] load balancer, the CRAB [27] load balancer, a filter from Suricata [44], a firewall from hXDP [5], and a bridge from Polycube [35]. We extend KLINT to analyze the assembly code compiled by the kernel after dumping it through a Linux debugging facility, using contracts we wrote for the BPF maps. Since we have no specifications for these BPF programs, KLINT only verifies memory safety and crash freedom. KLINT verifies the bridge and firewall in seconds, and CRAB and the Suricata filter in less than 2 minutes, but Katran requires almost 4 hours. This is due to our choice to model packet contents with a ghost map for simplicity, even though packet contents do not factor into invariant inference. Katran reads and writes to dozens of fields in the packet, which means the ghost map representing the packet has too many items to be efficient. This could be fixed by using a different way to model packet memory, such as Z3 arrays.

Developers can use any programming language, even if that language is considered too "exotic" or "immature" for operators to blindly trust it, since KLINT verifies binaries, and thus can catch errors resulting from compiler bugs.

For instance, the Rust [39] programming language is a promising direction for writing low-level systems code, including networking, but a developer or an operator might be concerned about its maturity level. Indeed, as of this writing there are currently 69 issues in the Rust bug tracker [46] marked with the "unsound" tag, meaning the compiler allows code that violates Rust's safety guarantees. Furthermore, some Rust features such as removing unused parts of the standard library are currently experimental.

However, we are able to write a traffic policer in Rust, compile it using these experimental features, and verify it using KLINT with the same specification as our C implementation. We can thus be confident that no matter what bugs lie in the Rust compiler, our binary is correct.

7 Limitations

KLINT's main limitation is that it cannot handle multi-threaded network functions that share state across threads. As we stated in §2, parallel code causes path explosion due to the amount of possible interleavings among threads. Running isolated instances of a network function in parallel can work by steering flows to cores, in which case KLINT can verify the instances, but this has skew issues.

KLINT imposes the following performance limitation on network functions: data structures must be dynamically linked so that KLINT can identify their operations by mapping the symbols left undefined in the binary to the contracts it is given as inputs. This can lead to slower function calls and lost opportunities for inlining.

KLINT also has the non-fundamental limitation of only supporting packet arrival as a trigger to run code and not timers or other events. Extending KLINT to support this is engineering work, replacing the restricted event handler model of packet reception with a more general one that branches on the event type.

Verifying network functions is only one part of the puzzle. KLINT can verify that a network function implements a protocol, but not that the protocol itself satisfies properties such as avoiding infinite message loops. Specialized techniques exist to verify protocols, such as IronFleet [22].

The remaining part of the puzzle is data structure implementations, for which manual verification tools currently remain necessary. We used VeriFast [25], which uses annotation for C, but other approaches exist such as Dafny [31], a programming language designed for verification.

8 Related work

Runtime verification is a related but distinct area of work, focusing on checking behavior at runtime. This catches bugs even in code currently beyond the reach of formal verification, for instance due to complex parallelism. Tools such as Aragog [52] trade completeness and performance for applicability. They only look at input and output packets and view network functions as "black boxes", thus they impose no constraints on code. But they cannot guarantee the absence of bugs, and in distributed environments cannot prevent bugs that can only be detected after compiling information from different machines. They also impose runtime overheads, unlike KLINT, due to the runtime nature of checks.

Symbolic execution is the main technique we build upon, and ghost maps may be useful beyond network functions. Symbolic execution is often used for bug finding instead of verification because of path explosion, but it can be augmented with techniques to bypass path explosion. Indeed, KLEE [7] and angr [48] were both designed primarily to find bugs, yet Vigor [53] and KLINT show that they can be used for verification. S2E [8] was also designed to find bugs but reused for network function verification by Dobrescu

and Argyraki [14]. Serval [40] is a symbolic execution engine enhanced with verification techniques and can prove systems such as a security monitor, at the cost of requiring some human annotations. KLINT could have used Serval as a base; we chose angr mostly because it is designed for quick prototyping. Path explosion can also be bypassed by writing code with few paths, as in the Hyperkernel [42].

Using maps as part of analyzing programs has been proposed before, though previous approaches were not aimed at functional verification, such as the Memsight [9] memory model for symbolic execution, or the technique proposed by Dillig et al. [13] to verify memory safety.

Network function verification tools such as Vigor [53], Gravel [55], and Prevail [21], which require source code, all inspired our design. Bolt [24] and Pix [23] verify performance instead of correctness, and require source code, though we believe they could use our techniques to only require binaries. While verifying binaries provides key advantages, verifying source code makes debugging failed verification easier since compiler optimizations can make it hard to match what the binary does wrong to what the code does wrong.

BPF is a more applicable but weaker form of network function verification: an in-kernel "verifier" checks memory safety and crash freedom, allowing untrusted code to run in kernel mode for performance without safety risks. BPF verifiers are fast, at the cost of restricting the code developers may write. BPF code cannot contain unbounded loops, must use specific data structures, and must include explicit checks for out-of-bounds memory accesses, even if a "smarter" and slower verifier might not need these checks. Internet infrastructure companies such as Cloudflare [32] use BPF as a core part of their infrastructure. Prevail [21] showed that formal methods can help BPF verification scale. Verified interpreters [51] and just-in-time compilers [41] for BPF exist, but they make no promises about functional correctness.

9 Conclusion

We presented KLINT, an automated tool to formally verify that network function binaries satisfy specifications with neither source code nor debugging symbols, given contracts for trusted data structures. This enables developers to provide guarantees about proprietary code to operators, removing a key barrier for adoption of formally verified network functions. KLINT uses maps as "universal" data structures for both specifications and contracts, enabling a sweeping simplification in reasoning including invariant inference. Using KLINT, we verified the functional correctness of 6 network function binaries written in C and 1 in Rust, and the memory safety and crash freedom of 5 existing BPF programs.

Acknowledgments

References

[1] ANGR CONTRIBUTORS. angr issue 938: SEGFAULT libz3. https://github.com/angr/angr/issues/938.

[2] BPF AUTHORS AND CONTRIBUTORS. bpf-helpers(7) Linux manual page. https://man7.org/linux/man-pages/man7/bpf-helpers.7.html.

[3] BRADLEY, A. R., MANNA, Z., AND SIPMA, H. B. What's decidable about arrays? In *Intl. Conf. on Verification, Model Checking, and Abstract Interpretation* (2006).

[4] BRADNER, S., AND MCQUAID, J. Benchmarking methodology for network interconnect devices. RFC 2544, RFC Editor, 1999.

[5] BRUNELLA, M. S., BELOCCHI, G., BONOLA, M., PONTARELLI, S., SIRACUSANO, G., BIANCHI, G., CAMMARANO, A., PALUMBO, A., PETRUCCI, L., AND BIFULCO, R. hxdp: Efficient software packet processing on FPGA NICs. In *Symp. on Operating Systems Design and Implementation (OSDI)* (2020).

[6] BRUNI, A., DISNEY, T., AND FLANAGAN, C. A peer architecture for lightweight symbolic execution. http://hoheinzollern.files.wordpress.com/2008/04/seer1.pdf, Unpublished.

[7] CADAR, C., DUNBAR, D., AND ENGLER, D. R. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symp. on Operating Systems Design and Implementation (OSDI)* (2008).

[8] CHIPOUNOV, V., GEORGESCU, V., ZAMFIR, C., AND CANDEA, G. Selective symbolic execution. In *Workshop on Hot Topics in Dependable Systems (HOTDEP)* (2009).

[9] COPPA, E., D'ELIA, D. C., AND DEMETRESCU, C. Rethinking pointer reasoning in symbolic execution. In *ACM Intl. Conf. on Automated Software Engineering (ASE)* (2017).

[10] CORBET, J. Bounded loops in BPF programs. https://lwn.net/Articles/773605/.

[11] CORBET, J. The BPF system call API, version 14. https://lwn.net/Articles/612878/.

[12] DE MOURA, L. M., AND BJØRNER, N. Z3: An efficient SMT solver. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (2008).

[13] DILLIG, I., DILLIG, T., AND AIKEN, A. Precise reasoning for programs using containers. In *ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)* (2011).

[14] DOBRESCU, M., AND ARGYRAKI, K. Software dataplane verification. In *Symp. on Networked Systems Design and Implementation (NSDI)* (2014).

[15] DPDK: Data plane development kit. https://dpdk.org.

[16] EISENBUD, D. E., YI, C., CONTAVALLI, C., SMITH, C., KONONOV, R., MANN-HIELSCHER, E., CILINGIROGLU, A., CHEYNEY, B., SHANG, W., AND HOSEIN, J. D. Maglev: A fast and reliable software network load balancer. In *Symp. on Networked Systems Design and Implementation (NSDI)* (2016).

[17] EMMERICH, P., GALLENMÜLLER, S., RAUMER, D., WOHLFART, F., AND CARLE, G. MoonGen: A scriptable high-speed packet generator. In *Internet Measurement Conf. (IMC)* (2015).

[18] EQUINIX. Network edge | Equinix edge services. https://www.equinix.se/services/edge-services/network-edge.

[19] FARSHIN, A., BARBETTE, T., ROOZBEH, A., MAGUIRE JR., G. Q., AND KOSTIĆ, D. PacketMill: Toward per-core 100-Gbps networking. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2021).

[20] FLANAGAN, C., AND LEINO, K. R. M. Houdini, an annotation assistant for ESC/Java. In *Intl. Symp. on Formal Methods Europe* (2001).

[21] GERSHUNI, E., AMIT, N., GURFINKEL, A., NARODYTSKA, N., NAVAS, J. A., RINETZKY, N., RYZHYK, L., AND SAGIV, M. Simple and precise static analysis of untrusted Linux kernel extensions. In *Intl. Conf. on Programming Language Design and Implementation (PLDI)* (2019).

[22] HAWBLITZEL, C., HOWELL, J., KAPRITSOS, M., LORCH, J., PARNO, B., ROBERTS, M. L., SETTY, S., AND ZILL, B. IronFleet: Proving practical distributed systems correct. In *ACM Symp. on Operating Systems Principles (SOSP)* (October 2015), ACM.

[23] IYER, R., ARGYRAKI, K., AND CANDEA, G. Performance interfaces for network functions. In *Symp. on Networked Systems Design and Implementation (NSDI)* (2022).

[24] IYER, R., PEDROSA, L., ZAOSTROVNYKH, A., PIRELLI, S., ARGYRAKI, K., AND CANDEA, G. Performance contracts for software network functions. In *Symp. on Networked Systems Design and Implementation (NSDI)* (2019).

[25] JACOBS, B., AND PIESSENS, F. The VeriFast program verifier, 2008.

[26] KAPUS, T., AND CADAR, C. A segmented memory model for symbolic execution. In *ACM SIGSOFT Intl. Symp. on the Foundations of Software Engineering (FSE)* (2019).

[27] KOGIAS, M., IYER, R., AND BUGNION, E. Bypassing the load balancer without regrets. In *Symp. on Cloud Computing (SOCC)* (2020).

[28] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The Click modular router. *ACM Transactions on Computer Systems (TOCS) 18*, 3 (2000).

[29] KUZNETSOV, V., KINDER, J., BUCUR, S., AND CANDEA, G. Efficient state merging in symbolic execution. In *Intl. Conf. on Programming Language Design and Implementation (PLDI)* (2012).

[30] LAN/MAN STANDARDS COMMITTEE. IEEE standard for local and metropolitan area networks: Media access control (MAC) bridges. Tech. rep., IEEE Standards Association, 2014. IEEE Std 802.1D-2004.

[31] LEINO, K. R. M. Dafny: An automatic program verifier for functional correctness. In *Intl. Conf. on Logic for Programming Artificial Intelligence and Reasoning (LPAR)* (2010).

[32] MAJKOWSKI, M. Cloudflare architecture and how BPF eats the world. https://blog.cloudflare.com/cloudflare-architecture-and-how-bpf-eats-the-world/.

[33] MCCANNE, S., AND JACOBSON, V. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Winter 1993 Conference* (San Diego, CA, Jan. 1993), USENIX Association.

[34] MERKEL, D. Docker: Lightweight Linux containers for consistent development and deployment. *Linux Journal* (2014).

[35] MIANO, S., BERTRONE, M., RISSO, F., BERNAL, M. V., LU, Y., PI, J., AND SHAIKH, A. A service-agnostic software framework for fast and efficient in-kernel network services. In *ACM/IEEE Symp. on Architectures for Networking and Communications Systems* (2019).

[36] MITRE CORPORATION. MS13-064. Available from CVE Details, CVE-ID MS13-064., 2013.

[37] MITRE CORPORATION. CVE-2014-9715. Available from CVE Details, CVE-ID CVE-2014-9715., 2014.

[38] MITRE CORPORATION. CVE-2015-6271. Available from CVE Details, CVE-ID CVE-2015-6271., 2015.

[39] MOZILLA RESEARCH. Rust programming language. https://www.rust-lang.org/.

[40] NELSON, L., BORNHOLT, J., GU, R., BAUMANN, A., TORLAK, E., AND WANG, X. Scaling symbolic evaluation for automated verification of systems code with Serval. In *ACM Symp. on Operating Systems Principles (SOSP)* (2019).

[41] NELSON, L., GEFFEN, J. V., TORLAK, E., AND WANG, X. Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the Linux kernel. In *Symp. on Operating Systems Design and Implementation (OSDI)* (2020).

[42] NELSON, L., SIGURBJARNARSON, H., ZHANG, K., JOHNSON, D., BORNHOLT, J., TORLAK, E., AND WANG, X. Hyperkernel: Push-button verification of an OS kernel. In *ACM Symp. on Operating Systems Principles (SOSP)* (2017).

[43] NETWORK WORKING GROUP. RFC 1812, requirements for IP version 4 routers. https://www.rfc-editor.org/rfc/rfc1812.txt, 1995.

[44] OPEN INFORMATION SECURITY FOUNDATION. Suricata website. https://suricata.io/.

[45] PIRELLI, S., AND CANDEA, G. A simpler and faster NIC driver model for network functions. In *Symp. on Operating Systems Design and*

Implementation (OSDI) (2020).

[46] Rust authors and collaborators. Issues - rust-lang/rust. https://github.com/rust-lang/rust/issues.

[47] Shirokov, N., and Dasineni, R. Open-sourcing Katran, a scalable network load balancer. https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer, May 2018.

[48] Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., and Vigna, G. SOK: (state of) the art of war: Offensive techniques in binary analysis. In *IEEE Symp. on Security and Privacy (S&P)* (2016).

[49] Turing, A. M. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society* (01 1937).

[50] Wang, J., Lévai, T., Li, Z., Vieira, M. A. M., Govindan, R., and Raghavan, B. Galleon: Reshaping the square peg of NFV, 2021.

[51] Wang, X., Lazar, D., Zeldovich, N., Chlipala, A., and Tatlock, Z. Jitk: A trustworthy in-kernel interpreter infrastructure. In *Symp. on Operating Systems Design and Implementation (OSDI)* (2014).

[52] Yaseen, N., Arzani, B., Beckett, R., Ciraci, S., and Liu, V. Aragog: Scalable runtime verification of shardable networked systems. In *Symp. on Operating Systems Design and Implementation (OSDI)* (2020).

[53] Zaostrovnykh, A., Pirelli, S., Iyer, R. R., Rizzo, M., Pedrosa, L., Argyraki, K. J., and Candea, G. Verifying software network functions with no verification expertise. In *ACM Symp. on Operating Systems Principles (SOSP)* (2019).

[54] Zaostrovnykh, A., Pirelli, S., Pedrosa, L., Argyraki, K., and Candea, G. A formally verified NAT. In *ACM SIGCOMM Conf. (SIGCOMM)* (2017).

[55] Zhang, K., Zhuo, D., Akella, A., Krishnamurthy, A., and Wang, X. Automated verification of customizable middlebox properties with Gravel. In *Symp. on Networked Systems Design and Implementation (NSDI)* (2020).

Appendix A Verified properties

We used Klint to verify both network functions we wrote and existing BPF network functions. We summarize the properties we proved for our network functions in this appendix. The BPF network functions have no formal specification and writing one would require a discussion with their authors to understand which edge cases are intended and which are not, so we instead chose to only verify that they are memory safe and free of crashes.

The full specifications are available on our repository: https://github.com/dslab-epfl/klint.

Bridge: we wrote a specification by manually extracting properties from the IEEE 802.1D [30] standard. For instance, section 7.7.1 of the standard, "Active topology enforcement", states that "Each Port is selected as a potential transmission Port if, and only if [...] The Port considered for transmission is not the Port on which the frame was received [...]", thus our specification checks that if the packet was transmitted, the transmission port must not be the reception port. As we explain in Section 6.1, we found a bug after translating section 7.8 of the standard, "The Learning Process", which states a condition for learning an address in the filtering database: "the source address field of the frame denotes a specific end station (i.e., is not a group address)".

Klint helps us write this specification by allowing us to write properties that must hold on packets and on state without having to explicitly depend on the bridge's internals. For instance, we do not need to specify the data type that the bridge uses to store metadata about Ethernet addresses, only that the bridge conceptually has a map with Ethernet addresses as keys, and that the source address of an incoming packet is or is not added depending on specification-related factors.

Firewall: beyond the partial specification of Listing 2, we wrote a full specification that is an evolution of the one from Vigor [53]. We model the firewall's state as a map from flows to last update time and ensure that the firewall (1) adds flows from the internal network to the state if possible, refreshing their last update time if necessary, and (2) only lets packets from the external network through if their flow belongs to the state. The specification also ensures that the firewall does not modify packet contents and does not drop packets unless they are incoming packets with no matching flow in the state.

Our specification is not concerned with how the firewall "remembers" flows, nor with when exactly this happens in packet processing, only that outgoing packets flows must be remembered if there is space and that incoming packets must be of a known flow if they are forwarded.

As we show in §6.1, Klint can also verify a more restricted specification that is only concerned with what happens when a packet is transmitted, including the fact that its flow must have been known previously.

Maglev: while we do not know of a formal specification for Maglev, our specification is an evolution of the one from Vigor [53], which was written according to the behavior described by the Google paper [16]. We model the state as two maps, one from flows to backends and one from backends to last heartbeat time. Packets from backends are heartbeats and must update the corresponding last heartbeat time and then be dropped. Packets from clients must be routed to a backend and must not be dropped unless there are no available backends.

NAT: our specification is an evolution of the one from VigNAT [54], which fully describes the behavior of the NAT. We model the state as a map from flows to last update time, in a similar fashion to the firewall specification. Packets must only be dropped if they are not IP or TCP/UDP, as the NAT does not support other protocols (for now), or if they come from the external network but do not belong to a known flow. Packet headers must be updated according to the state, and packets from the internal network must trigger state updates.

One interesting aspect of Klint with regards to the NAT is invariant inference: Vigor's original NAT required about 3000 lines of manually written proof annotations for the invariants between the three data structures the NAT uses internally, and the Vigor specification was dependent on

these data structures. KLINT instead infers these invariants automatically, and our specification defines the NAT in terms of maps.

Policer: our specification is an evolution of the one from Vigor [53], which fully describes the behavior of the policer. We model the state as one map from buckets to tokens, and one map from IP addresses to buckets. The specification then enforces that the policer must update the state according to incoming packets and their size and drop packets if their bucket is too full.

Router: we wrote a specification based on RFC 1812 [43], "Requirements for IP Version 4 Routers". We model the state as a map from CIDR blocks to devices. We enforce the header validation requirements from section 5.2.2 of the RFC, the time to live requirements from section 4.9.9.2, and most importantly the longest-prefix-match property to find the next hop address from section 5.2.4.3.

Appendix B   Ghost maps *get* algorithm

We present here the algorithm for the *get* operation on ghost maps. PC is the Path Constraint. UK_M, UV_M, and UP_M are the triple forming map M's unknown item, as we explain in §3.3. The *condition* and *value_hint* are those required to handle invariant recursion, as we explain in §3.4.

function KnownSize(M)
    $result = 0$, $known = \emptyset$
    for (k, v, p) in M's *known items* **do**
       $result \mathrel{+}= ITE(k \notin known \wedge p, 1, 0)$
       $known \mathrel{+}= k$
    return $result$

function Get(M, K)
    for (k, v, p) in M's *known items* **do**
       if *unsatisfiable*$(condition \wedge K \neq k)$ **then**
          return (v, p)
    if *unsatisfiable*$(condition \wedge K \neq UK_M)$ **then**
       return (UV_M, UP_M)
    Let (V, P) be a fresh value and presence bit
    if *condition* is set **then**
       Add *condition* $\Rightarrow V = value\_hint$ to the PC
    Let $U = \bigwedge (K \neq K')$ for each known key K' in M
    Add (K, V, P) to M's known items
    for (k, v, p) in M's *items* **do**
       Add $K = k \Rightarrow \big((V = v) \wedge (P = p)\big)$ to the PC
    Add $U \Rightarrow invariant_M(K, V, P)$ to the PC
    Add $KnownSize(M) \leq length(M)$ to the PC
    return (V, P)

Differential Network Analysis

Peng Zhang*, Aaron Gember-Jacobson♯, Yueshang Zuo*, Yuhao Huang*, Xu Liu*, and Hao Li*
*Xi'an Jiaotong University, ♯Colgate University

Abstract

Networks are constantly changing. To avoid outages, operators need to know whether prospective changes in a network's control plane will cause undesired changes in end-to-end forwarding behavior. For example, which pairs of end hosts are reachable before a configuration change but unreachable after the change? Control plane verifiers are ill-suited for answering such questions because they operate on a single snapshot to check its "compliance" with "explicitly specified" properties, instead of quantifying the "differences" in "affected" end-to-end forwarding behaviors. We argue for a new control plane analysis paradigm that makes differences first class citizens. Differential Network Analysis (DNA) takes control plane changes, incrementally computes control and data plane state, and outputs consequent differences in end-to-end behavior. We break the computation into three stages—control plane simulation, data plane modeling, and property checking—and leverage differential dataflow programming frameworks, incremental data plane verification, and customized graph algorithms, respectively, to make each stage incremental. Evaluations using both real and synthetic control plane changes demonstrate that DNA can compute the resulting differences in reachability in a few seconds—up to 3 orders of magnitude faster than state-of-the-art control plane verifiers.

1 Introduction

Networks are frequently in flux. Configurations are modified monthly, or even weekly: e.g., two large universities change up to 55 stanzas per router per month [27], and Facebook conducts an average of 12.5 changes per device per week in their backbone [41]. External peers update routes daily: e.g., four tier-1 ISPs each experience a median of ∼100K route updates per day [12]. Links and routers fail intermittently: e.g., a large online service provider's data centers have a median of 18.5 link outages per day [17], and the CENIC research network has a median of 38.5 outages per link per year [42].

Each change poses a risk of introducing catastrophic network outages [28, 31, 46]. To avoid outages, operators need

to know whether prospective changes in the control plane (i.e., changes in configurations, external routes, or available links/routers) will cause (un)desired changes in end-to-end forwarding behavior. For example, would a link failure break isolation? Would a configuration change reduce reachability? Would an external route withdrawal degrade load balancing?

At first glance, existing verifiers [1, 4, 6, 7, 13–15, 18, 22–24, 26, 33, 37, 40, 43, 45, 48] seem to address this need. However, data plane verifiers [18, 19, 23, 24, 26, 33, 44, 48] cannot directly answer these questions, because they operate on changes in the control plane's output, rather than the control plane itself. Existing control plane verifiers [1, 4, 6, 7, 13–15, 22, 37, 40, 43, 45] are also ill-suited for this task, because of the following limitations: (1) they focus on checking a single control plane snapshot instead of differences between snapshots and (2) they require a list of properties to check, but it is difficult for operators to determine which properties may be affected by a change and hence need to be checked.

For the first limitation, a possible workaround is to apply a control plane verifier to both the old and new snapshots and compare the verifier's output. But analyzing both snapshots from scratch is wasteful, because many control plane changes have a limited impact on the data plane. For example, fewer than 100 forwarding (ACL) rules are changed for >80% (>90%) of configuration changes in the backbone network at a large university (Figure 1), and Steffen et al.'s experiments with 90 ISP topologies show that only one-third of link failures impact the forwarding path between a randomly chosen ingress node and destination prefix [40].

For the second limitation, a possible workaround is to check reachability (or other properties) for all pairs of end hosts. But checking all possible properties is wasteful, because the space of all properties is large [9] and the set of properties affected by a change is often small. Policy-mining tools [8, 9, 25] can help narrow the space of properties, but the set of inferred properties may still be large—e.g., Config2Spec returns over 3K properties for a national research and education (R&E) network with only 10 routers [9].

Since control plane verifiers can be quite inefficient for

assessing whether control plane changes cause changes in end-to-end behavior, we argue for a new control plane analysis paradigm which makes differences first class citizens. *Differential Network Analysis (DNA)* takes as input differences in configurations, external routes, and available links/routers, incrementally computes control and data plane state, and outputs the consequent differences in end-to-end behavior (e.g., reachability, waypointing, load balancing, etc.). This aligns with the small size/impact of many control plane changes and avoids duplicate and unnecessary computations.

To incrementally compute differences in forwarding behavior based on differences in the control plane, DNA breaks the computation into three modular stages—control plane simulation, data plane modeling, and property checking—and makes each stage "differential". In other words, each stage consumes differences (control plane changes, data plane changes, and forwarding graph changes, respectively), incrementally updates its network model, and produces differences (data plane changes, forwarding graph changes, and property changes, respectively). To achieve incremental computation for each of these stage, DNA leverages differential dataflow programming frameworks [2, 35], incremental data plane verification [48], and customized graph algorithms, respectively.

We implement a version of DNA that supports differential analysis of two widely-used routing protocols (BGP and OSPF) and widely important properties (reachability, waypointing, and load balancing). Our implementation of DNA is publicly released under an open source license. We evaluate DNA using both synthetic and real control plane changes, and demonstrate that DNA can compute differences in reachability induced by control plane changes in a few second—up to 3 orders of magnitude faster than state-of-the-art control plane verifiers [1,4,6]. Second-level verification time can enable on-the-fly checking of operator-proposed configuration changes, similar to syntax checkers integrated into most programming IDEs, as well as quickly validating automatically-generated changes due to dynamic control [30].

In summary, we make the following three contributions:

- We propose differential network analysis (DNA), a new paradigm that helps operators better understand the impact of changes in the control plane.
- We design and implement DNA based on recent advances in differential dataflow programming and incremental data plane verification, and apply optimizations to overcome their inefficiencies.
- We use both synthetic and real control plane changes to show DNA computes consequent differences in end-to-end behavior up to 3 orders of magnitude faster than state-of-the-art control plane verifiers [1,4,6].

2 Motivation

In this section, we discuss in detail why invoking a control plane verifier before and after a change, and for all inferred
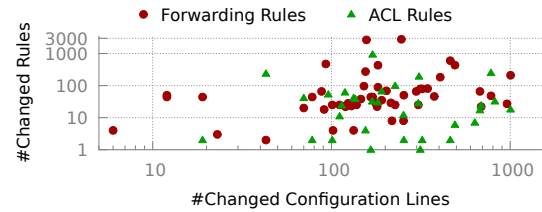

Figure 1: The number of changed (forwarding and ACL) rules, and the number of changed configuration lines for the backbone network at a large university.

properties, is an inefficient way to assess whether changes in the control plane cause changes in end-to-end behavior. In particular, we highlight: (1) the prevalence of small control plane changes, and (2) the difficulty of identifying which properties may be impacted by a change.

2.1 Control plane changes are often small

Control plane verifiers operate on full snapshots of the control plane, but the delta between snapshots is often small, which leads to significant amounts of unnecessary re-computation.

Configuration changes. Several prior studies have examined router configurations across various organizations/networks and found that changes are often small: e.g., two large universities each change (on average) ≤ 20 lines of configuration at the same time [36]; changes in Facebook's backbone and data center networks impact an average of 157 and 738 lines of configuration, respectively, which is relatively small compared to the scale of these networks [41]; and in 75% of the networks operated by a large online service provider, the median change includes only three devices [16]. Consequently, we expect control plane verifiers' inputs and outputs to be similar before and after such changes.

To validate this hypothesis, we analyze 3 months of configuration changes from the backbone network at a large university [36]. The network has 28 routers and 50 links and runs OSPF. On average, the network has ∼75K total lines of configuration, which generate ∼25K total forwarding rules. The configuration changes include adding/removing subnets, access control lists (ACLs), OSPF routes, etc. Figure 1 depicts the size of each configuration change and the corresponding number of changes in forwarding rules and ACL rules. On average, 228 lines of configuration (∼0.3%) are changed, causing 146 forwarding rules (∼0.6%) and 34 ACL rules (∼0.9%) to change. Except for two updates, all configuration changes result in <600 changes in forwarding rules.

External route changes. Prior studies have shown that autonomous systems (ASes) may experience a high rate of BGP updates—e.g., four tier-1 ISPs each experience a median of ∼100K route updates per day [12]—yet only 1% (10%) of next-hops in the Internet change each day (month) [10]. Con-

sequently, we expect external route changes to have a small impact on a network's control and data planes.

To validate this hypothesis, we analyze 1 year of hourly RIB snapshots from a national R&E network [3]. We exclude hours in which a configuration change was made to ensure we only capture RIB changes caused by external route changes. We find <15% of hours have at least one RIB entry change, and <4% of hours have more than 10 RIB entries change.

Link/router availability changes. Link/router failures—caused by software upgrades, hardware faults, etc.—are common: e.g., the CENIC Digital California and High Performance Research networks experience a median of 5.1 and 38.5 failures per link per year, respectively [42], and tens of geographically distributed data centers operated by a large online service provider experience a median of 18.5 link failures and 3 device failures per day [17]. However, Steffen et al.'s experiments with 90 ISP topologies showed that for a randomly chosen ingress node and destination prefix, two-thirds of link failures do not impact the forwarding path from the ingress to the destination [40]. In other words, we expect only a fraction of FIB entries to change when links/routers fail.

2.2 Identifying behaviors to (re-)verify is hard

Identifying how changes in the control plane impact end-to-end forwarding behavior is important for assessing whether the changes: (1) have the intended effect, and (2) have any undesirable side-effects.

For the former, it is easy to identify which behaviors to examine, because these behaviors are effectively the "design requirements" for the change. For example, operators may propose a change in filters to restrict access to a new subnet; the effectiveness of the change can be assessed by examining reachability between the restricted and new subnet(s).

In contrast, it is difficult to identify which behaviors should be examined to determine whether a change is "safe." Some behaviors may be obvious, because they relate directly to the change: e.g., to assess the safety of the proposed change in route filters, operators should examine reachability from non-restricted subnets to the new subnet. However, a change can also impact seemingly unrelated behaviors—e.g., reachability from non-restricted subnets to existing subnets—or behaviors outside of an operator's purview—e.g., changes in a single data center may affect how a WAN load balances traffic across multiple data centers. The latter can arise especially when different teams manage different aspects of a network, networks are merged (e.g., due to an acquisition), or operators with historical knowledge of the network leave an organization.

A simple way to ensure a change does not negatively impact seemingly unrelated behaviors is to examine all categories of end-to-end behaviors for all (pairs of) prefixes. However, this is prohibitively expensive [9], and likely results in lots of unnecessary computation—e.g., if access from a subnet was already restricted, then further restrictions in access are unlikely to impact reachability. Ideally, only behaviors that could potentially be impacted should be examined.

In summary, repeatedly analyzing full snapshots of the control plane is wasteful, and determining which end-to-end behaviors to analyze is hard.

3 Overview

This section overviews DNA, a modular network analysis framework which can incrementally compute the "differences" in forwarding behavior that arise from "differences" in the control plane. We will first present the framework of DNA, and use an example to show its workflow. After that, we discuss three challenges when realizing DNA.

3.1 The DNA workflow

DNA is a modular framework with three stages, where each stage consumes and produces some forms of *differences*.
- The first stage consumes control plane changes and simulates the control plane to generate differences in data plane state (i.e., insertions/deletions of forwarding rules).
- The second stage updates a data plane model to generate differences in forwarding graphs (i.e., insertions/deletions of packet equivalence classes on edges).
- The third stage identifies and checks relevant end-to-end forwarding behaviors to generate differences in end-to-end properties (e.g., changes in reachability).

We use an example to show how these three stages work.

An example network. Figure 2(a) shows an example network which is used throughout the paper. The network has five routers running BGP. Router E announces two /24 prefixes and one /16 prefix. There is an outbound route filter at port 2 of router E, which filters routes for the /16 prefix, and an inbound ACL at port 2 of router D, which drops all traffic to the two /24 prefixes. The green and yellow arrows represent the best routes selected at each router.

To simulate a change, we consider a single link failure, i.e., the link between router C and E fails. As a result, router A can no longer reach the two /24 networks. We show how DNA can uncover this change in reachability.

Stage 1. Differential control plane simulation (§4). This stage takes as input differences in control plane state, incrementally simulates the control plane, and outputs differences in data plane state. Differences in control plane state are insertions/deletions of configuration lines, external routes, or links/routers, and differences in data plane state are insertions/deletions of forwarding/ACL rules. Generally, simulating a control plane entails modeling the route propagation, filtering, and selection behaviors within and across distributed routing protocols to compute the converged control plane state and produce a concrete data plane [1, 7, 14, 34, 37, 38]. To be incremental, we must transition from one converged state to
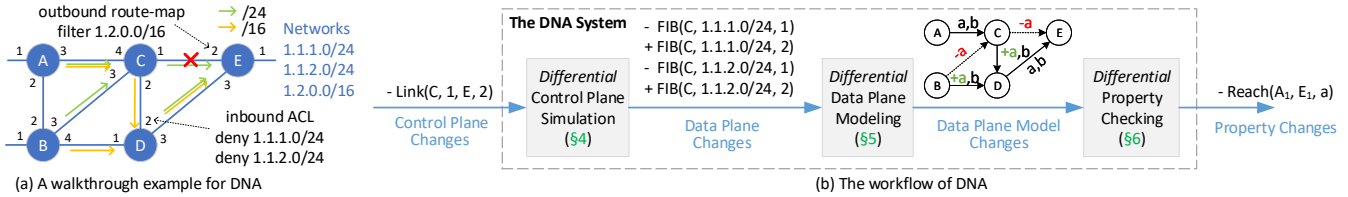
Figure 2: An example demonstrating the workflow of DNA.

another, accounting for the impact of control plane changes on route propagation, filtering, and/or selection across routers.

In our example, the link failure between port 1 of C and port 2 of E will be encoded as a deletion of a record $Link(C, 1, E, 2)$, which is the input the Stage 1. Given this input, DNA simulates the control plane and generates FIB differences: e.g., at router C, the two /24 routes whose output port is 1 (2) are deleted (inserted), as shown Figure 2.

Stage 2. Differential data plane modeling (§5). This stage takes as input differences in data plane state, incrementally constructs a data plane model, and outputs differences in the data plane model. Generally, a data plane model partitions the packet space into *Equivalence Classes (ECs)*, each of which represents a set of packets with the same forwarding behavior through the network [26]. Then, the model concisely encodes the forwarding behavior of packets with a forwarding graph, where each edge is labelled with ECs that can traverse it [18, 48], as shown in Figure 2. When rules are inserted or deleted, ECs are split, merged, and transferred among edges, to reflect the forwarding behavior change.

In our example, there are three ECs in total, where a represents $[1.1.1.0, 1.1.2.255]$, b represents $[1.2.0.0, 1.2.255.255]$, and c represents all other IP addresses (which are not shown here). Since forwarding rules for the two /24 prefixes which output to port 1 are deleted, and those which output to port 2 are inserted, EC a "transfers" from edge (C, E) to edge (C, D) and from edge (B, C) to edge (B, D), as shown in Figure 2. Thus, the differences of data plane model consist of two insertions and two deletions for EC a on these four edges.

Stage 3. Differential property checking (§6). This stage takes as input differences in the data plane model (i.e., insertions and deletions of ECs on the forwarding graph), incrementally computes relevant forwarding behaviors, and outputs the differences in properties. Here, the differences in properties, which we term as *differential properties*, are defined as insertions and deletions of forwarding properties, including reachability, waypointing, load balancing, etc. DNA computes differential properties by traversing the forwarding graphs from edge ports, which are ports that connected to the hosts, servers, or other networks. The traversal starts with a set of all affected ECs, which are updated by intersecting with those ECs labelled on the edge. The traversal ends when the set of ECs becomes empty or another edge port is reached.

In our example, we have three edge ports, i.e., port 1 of A, B, and E. By traversing from these edge ports, we get the

differential reachability as $-Reach(A_1, E_1, a)$, meaning that packets belonging to EC a can no longer reach port 1 of E from port 1 of A. As will be shown in §6.2, DNA optimizes the above computation by traversing directly from the *change points*, where network forwarding behaviors change (B and C in this example). Note that network invariants like loop-freedom and blackhole-freedom are covered by differential reachability, since loops or blackholes will result in some pairs of end points becoming unreachable.

3.2 Challenges in realizing DNA

Realizing each of the three stages of DNA requires addressing the following three challenges, respectively.

(1) How to achieve control plane simulation in a way that is easy to extend? Network control plane has complex semantics, and simulating a control plane to cover all relevant feature often lead to complex code base. Generally, an incremental algorithm can be much more complex than its non-incremental counterpart [39]. Thus, incremental control plane simulation can be difficult to build. In addition, this is not a once-for-all task, considering the semantics of the control plane are still evolving, and new vendor-specific features emerge. Therefore, DNA needs to realize incremental control plane simulation in an easy-to-extend way.

(2) How to efficiently update data plane model for batched rule updates? Existing data plane verifiers are designed to consume single-rule updates—i.e., for each rule insertion or deletion, they individually update the model. For DNA, however, the differences in data plane state consist of a batch of rule updates. As an example, a link failure causes deletions of rules for multiple prefixes which are previously forwarded through the failed link. In such a setting, consuming single-rule updates can result in redundant computation, since these rule updates are often correlated. Therefore, DNA needs to optimize the data plane model update algorithm for batched rule updates.

(3) How to determine and only re-check affected properties? Existing data plane verifiers can incrementally check invariants like blackhole-freedom and loop-freedom by only re-verifying the ECs affected by a FIB update. DNA, however, needs to check all properties of some type (e.g., all-pairs reachability), and re-checking all these properties is wasteful since often only a small portion of the properties are affected. Therefore, DNA needs a way to determine which properties

are affected, which is not easy: e.g., it is unclear which reachability properties may be affected by a change in OSPF cost.

We describe how we address these challenges in §4–§6.

4 Differential Control Plane Simulation

As the first stage, DNA maps differences in configurations, external routes, and available links/routers to differences in FIBs. In the following, we show how DNA achieves this mapping in an incremental and easy-to-extend manner.

4.1 Modeling the control plane

A network control plane computes routes in a recursive way: each router receives routes from neighboring routers, filters and/or modifies the routes, locally ranks the routes to select the best routes, and advertises the best routes (which may be filtered/modified) to neighboring routers. The steps are repeated until routing converges—i.e., no more changes are made to any routing information bases (RIBs).

Leveraging differential dataflow for automatic differential computation. The aforementioned process fits well into the *dataflow* programming model. A dataflow program corresponds to a directed graph where vertices represent operators (i.e., functions that transform data), and edges represent the flow of data between operators [20]. For control planes, filtering, modifying, and selecting routes can be modeled with operators, and the propagation of routes between RIBs corresponds to the flow of data.

Differential Dataflow (DD) [35] is one dataflow programming framework which supports general incremental computation for recursive dataflows. This is achieved with a set of differential operators like `join`, `count`, etc. which efficiently produce differences in outputs from differences in inputs.

The left of Figure 3 shows a part of the dataflow program for OSPF route propagation using operators offered by DD. It `joins` the routes stored in a collection `BestOSPFRoute` with another collection `OSPFNeighbor` to model the propagation of routes from one router to another router; `filters` the routes where the origin of the route is the router itself (in order to prevent loops); `joins` with the collection `OSPFCost` to get the OSPF cost configured on the interface where the route is received; `maps` the routes to new routes with the OSPF cost updated by adding the interface's cost; and finally `joins` with `InterfaceIP` to get the IP of the interface to produce routes `OSPFRoute`, which will be further be processed to produce the `BestOSPFRoute` (omitted here).

Leveraging differential Datalog for better extensibility. As can be seen in the above example, modeling route computation directly using low-level operators offered by DD is less intuitive and can take a lot of effort, making the model hard to extend to new (vendor-specific) protocols or features. Therefore, we leverage *Differential Datalog (DDlog)* [2], a

Datalog programming language built on top of DD. In the following, we give some preliminary to Datalog, and introduce our DDlog-based control plane model.

A Datalog program consists of a set of *facts* and *rules*. A fact is a statement like "interface *intf* of router *X* has OSPF cost *cost*". This fact can be represented with `OSPFCost`$(X, intf, cost)$, where `OSPFCost` is termed a *relation*. A rule takes the form of $R_1(u_1) : -R_2(u_2), \ldots, R_n(u_n)$, where each R_i is a relation, meaning $R_1(u_1)$ holds if $R_2(u_2), \ldots, R_n(u_n)$ hold. Given some *base facts*, one can derive new facts by firing the Datalog rules.

DDlog-based control plane model. The right of Figure 3 shows the corresponding DDlog rule for the corresponding dataflow model on the left. As we can see, DDlog allows us to only focus on how routes (i.e., facts) are derived, without caring about the sequence to join, map, or filter routes. Additionally, unlike other Datalog languages [5, 21], DDlog offers several useful data structures, such as vectors, which make the modeling of route computation much easier.

Figure 4 shows the flow of data in our DDlog-based control plane model. If relations *A* and *B* appear on the left and right side of a rule, respectively, then there is an edge from *B* to *A* in the graph. If multiple relations appear on the right side of a rule to derive relation *A*, we merge their edges to *A*.

There are three types of relations: input relations, output relations, and intermediate relations. The input relations contain base facts including: (1) configurations for routing protocols, e.g., `BGPNet` contains the subnets imported to BGP; (2) network topology, e.g., `Link` contains L3 links; and (3) external routes, e.g., `ExtRoute` contains routes announced by ISPs that are out of scope of our model. There is a single output relation, i.e., `FIB`, and multiple intermediate relations, e.g., `GlobalRIB` which contain derived facts, e.g., routes.

The DDlog-based control plane model treats changes in the control plane as insertions and deletions of facts in input relations. For example, modifying the OSPF cost on interface 3 of router *D* from 10 to 100, is treated as two changes $-$`OSPFCost`$(D, 3, 10)$ and $+$`OSPFCost`$(D, 3, 100)$. The resultant changes in data plane state are insertions and deletions of facts in the output relation, i.e., `FIB`.

4.2 Executing the control plane model

The control plane model, which is a DDlog program, will be compiled into a DD program for execution. When executing the DD program, changes are propagated in the dataflow graph, and at each operator the changes in input will be mapped to changes in output. Since the dataflow is recursive, the propagation continues for multiple iterations, until there are no more changes (fixed point is reached). The insertions and deletions in the output relation `FIB` will be returned, and will be fed to Stage 2 (§5).

Figure 5 shows the execution of the control plane model for our example network. We have over-simplified the execution
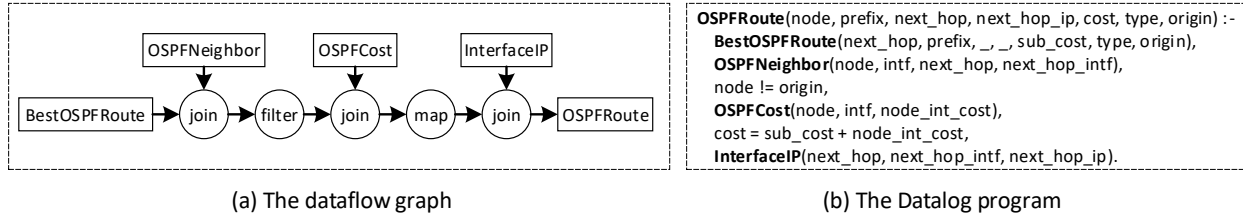
(a) The dataflow graph

(b) The Datalog program

Figure 3: Part of dataflow graph with the corresponding Datalog program snippet for OSPF route propagation.
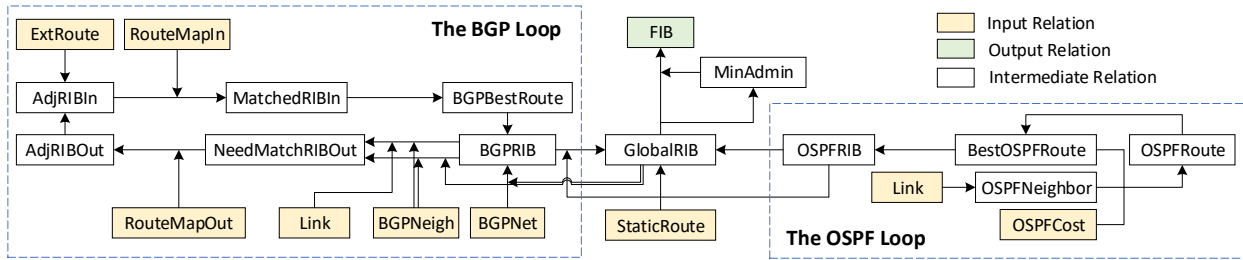

Figure 4: The flow of data in our control plane model. For simplicity, only the core relations are shown.
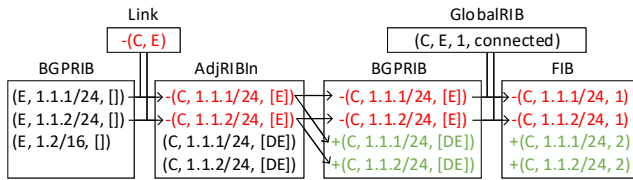

Figure 5: Control plane simulation for the example network.

by removing changes to a lot of intermediate relations and only focusing on changes to `Link`, `AdjRIBIn`, `BGPRIB`, and `FIB`. Here, the link failure $-(C,E)$ is input to the DD computation engine. The deletion will be `join`ed with existing facts in `BGPRIB` to derive new facts. Since the change is a deletion so the changes it derives are also deletions. After multiple intermediate steps, the change derives $-(C, 1.1.1/24, [E])$ and $-(C, 1.1.2/24, [E])$ in `AdjRIBIn`, meaning deletions of the received routes for prefixes 1.1.1.0/24 and 1.1.2.0/24 with AS path $[E]$ at router C. These two deletions in `AdjRIBIn` will trigger deletions of the old best routes and insertions of the new best routes for router C in `BGPRIB`. The changes in `BGPRIB` will then be joined with `GlobalRIB` to generate four changes in the output relation `FIB`. For simplicity, only the join of two deletions in `BGPRIB` and the corresponding fact in `GlobalRIB` are shown in Figure 5.

4.3 Optimizations

Customizing functions for efficiency. As noted above, using operators like `join` and `map` in DD can express simple operations like route propagation (routes are sent to neighboring routers, costs are updated, etc.). However, directly modeling more complex operations—e.g., BGP best route selection, applying route policies, etc.—using these DD operators requires a lot of operators, making the evaluation less efficient.

For example, suppose we select best routes from received routes R based on two conditions: local preference (LocPref) and path length (PathLen). Using DD, we need to group routes in R by prefix and use the aggregation function `max` to compute the highest LocPref value for each prefix, and then `join` the resultant collection $R1$ with R to obtain another collection $R2$ which contains routes with the highest LocPref. Similarly, we need to compute another two collections for PathLen. Correspondingly, if we use DDlog, we need to declare two rules and two relations for each condition. The original Datalog-based version of Batfish [14] used the above approach to realize BGP best route selection. Since there are many criteria for BGP best route selection (Cisco uses 13 criteria), we need a lot of DD operators, or correspondingly a lot of DDlog rules, making the model inefficient to evaluate.

To make the model efficient to evaluate, we realize complex operations (e.g., best route selection and route policies) with customized functions, which can be wrapped inside a `Reduce` operator offered by DD. Appendix C gives the code snippet of the function for best routes selection. In our experiments, by just customizing the process of best route selection, we can achieve a ~40% speedup (§8.1).

Partitioning routes for parallel simulation. For the same routing protocol, the propagation of different routes (prefixes) are largely independent. In the absence of route aggregations, two BGP routes 1.1.1.0/24 and 1.1.2.0/24 propagate independently through the network. Therefore, we partition the routes of the same routing protocol into groups, for parallel evaluation, and merge their results to obtain the RIBs for this protocol. In the following, we discuss why this works when there are multiple protocols and route aggregations.

(1) Route Dependency. When there are multiple routing protocols, routes may have dependencies. For example, BGP routes may depend on the OSPF routes for the loopback inter-
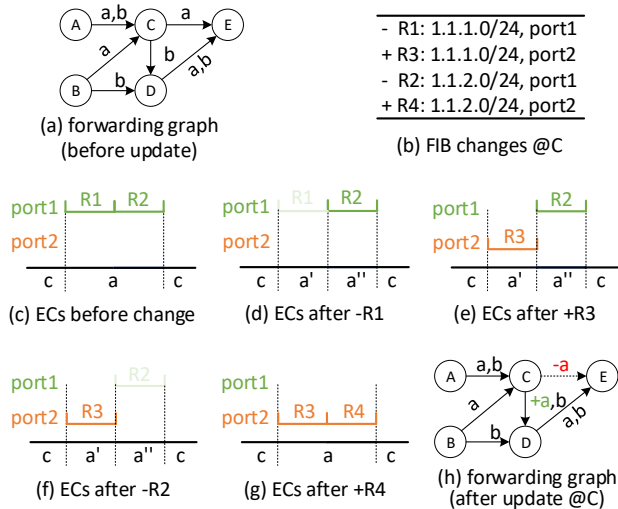
(a) forwarding graph (before update)	- R1: 1.1.1.0/24, port1 + R3: 1.1.1.0/24, port2 - R2: 1.1.2.0/24, port1 + R4: 1.1.2.0/24, port2 (b) FIB changes @C

(c) ECs before change   (d) ECs after -R1   (e) ECs after +R3

(f) ECs after -R2   (g) ECs after +R4   (h) forwarding graph (after update @C)

Figure 6: Data plane model update for the example network.

faces of all iBGP peers in the same AS. We adopt a simple approach where we group BGP routes and OSPF routes separately, and schedule BGP groups only after all OSPF groups are finished. More sophisticated scheduling [37] can be used to handle more complex dependencies.

(2) Route Aggregation. Sometimes, different routes may be correlated due to `route aggregation`. For example, a router can aggregate routes for 1.1.1.0/24 and 1.1.2.0/24 into a single route (1.1.0.0/16) when advertising to its neighbors. Although it may seem this correlation prevents routes for 1.1.1.0/24 and 1.1.2.0/24 from being computed separately, our method is not affected since each instance has all the rules and base facts describing route aggregation. Thus, even if these two routes are in different groups, both of them can be aggregated into 1.1.0.0/16. We can remove the duplicated routes of 1.1.0.0/16 when merging the routes of multiple instances.

5 Differential Data Plane Modeling

As the second stage, DNA maps differences in data plane state to differences in the data plane model. We accomplish this using APKeep [48], a state-of-the-art data plane model.

In the following, we first show that updating the model by treating each data plane update separately can result in redundant computation, and then show how DNA can leverage correlation among rule updates to reduce such redundancy.

5.1 Single-rule model update

We return to the example network to show how APKeep updates the data plane model. We only consider the rule updates at Router C, as shown in Figure 6(b). The rule updates at Router B are quite similar and thus not discussed here.

Step 1. Identifying forwarding behavior changes. For each rule update, APKeep identifies the packets that change for-

warding behavior[1] by analyzing rule dependency. Returning to the example, after removing $R1$, APKeep determines that packets which previously match destination IP addresses in 1.1.1.0/24 will not match any lower-priority rule, and thus will be dropped. Then, the forwarding behavior change will be a 3-tuple $(1.1.1.0/24, port1, drop)$ which specifies the affected packets, old, and new output port, respectively.

Step 2. Updating the forwarding graph. For each change, APKeep updates the ECs, and transfers the updated ECs on the forwarding graph. Specifically, APKeep iterates over all ECs assigned to the old port, and check whether each EC belongs to or intersects with the affected packets. For the former, the EC will be transferred directly; while for the latter, the EC needs to be split before the transfer. In this example, the affected packets 1.1.1.0/24 will split EC a into two ECs, a' for 1.1.1.0/24, and a'' for 1.1.2.0/24, as shown in Figure 6(d). Then, a' will be transferred from port 1 to port $drop$ (a default port not shown here).

Figure 6(e) shows the insertion of $R3$, where step 1 identifies a behavior change $(1.1.1.0/24, drop, port2)$, and step 2 transfers a' from port $drop$ to port 2. Figure 6(f) shows the deletion of $R2$, and Figure 6(g) shows the insertion of $R4$, after which EC a' and EC a'' have the same forwarding behavior and are merged into a single EC a. Figure 6(h) shows the resulting differences of data plane model which are insertions/deletions of EC a on two edges.

In large networks, a configuration change may produce hundreds or thousands of rule updates, and performing the above two steps for each of them is slow. For example, failing a link in a fat tree with 180 routers results in over 3K rule updates. Even though a single rule update takes only 1ms, it still amounts to 3 seconds.

5.2 Batched model update

We observe that even when there are many rule updates, they are highly correlated, such that we can batch them to reduce redundant computation. In the following, we consider two types of correlations.

Correlation among rule insertions and deletions. Rule deletions are often accompanied by rule insertions for the same IP prefix. Many configuration changes like changing a BGP local preference or OSPF link cost will make the router change the best routes for some destination prefixes. Each change of best route would translate into a deletion of the old rule and an insertion of a new rule.

Based on the above correlation, we can *batch rule deletions and insertions* for step 1. Returning to our example, deleting $R1$ and inserting $R3$ requires updating the data plane model twice. However, by batching the deletion of $R1$ and insertion of $R3$, we can directly identify the change

---

[1] In this section "forwarding behavior" refers to hop-by-hop forwarding, not end-to-end forwarding.

as $(1.1.1.0/24, port1, port2)$. Therefore, we only need to run step 1 once. Moreover, step 1 will be more efficient since we do not need to analyze rule dependency. Similarly, by batching the deletion of $R2$ and insertion of $R4$, we can use another run of step 1 to identify a change $(1.1.2.0/24, port1, port2)$.

Correlation among rule updates on the same device. Rule updates on the same device are often quite similar, e.g., route deletions/insertions have the same output port. Many configuration changes like bringing down/up an interface will delete routes that output to the interface, and add new routes that output to other interfaces.

Based on the above correlation, we can *batch forwarding behavior changes on the same device* for step 2. Returning to the example, the two changes $(1.1.1.0/24, port1, port2)$ and $(1.1.2.0/24, port1, port2)$ have the same old port and the same new port. Instead of performing step 2 for each of these two changes, we can batch them as a single one $(1.1.1.0/24 \vee 1.1.2.0/24, port1, port2)$, and run step 2 only once. Moreover, we can directly transfer EC a from $port1$ to $port2$, without splitting a, further reducing computation overhead. Since step 2 needs to check all ECs of the old port, each involving a BDD operation, it dominates the overall running time of model update, and by batching changes for step 2, we can significantly reduce the overall running time.

In sum, APKeep needs to run the above two steps for each of the four rule updates, while after batching DNA only needs to run step 1 twice (without analyzing rule dependency), and step 2 once (without splitting and merging of ECs). As a result, DNA can directly update the model as Figure 6(g), avoiding intermediate steps shown in Figure 6(d)-(f).

Note that some of the batching methods can also be applied to other data plane verifiers. For example, Delta-net [18] can also be modified to leverage the first correlation. However, it is not clear how Delta-net can leverage the second correlation.

6 Differential Property Checking

This stage tracks network properties and returns differences, which we call *differential properties*. We focus on differential reachability, differential waypointing, and differential load balancing. In this section, we define these differential properties, and introduce an algorithm to efficiently compute them.

6.1 Defining differential properties

A network can be viewed as a big switch providing connectivity among entities including hosts, servers, middleboxes, external networks, etc. We term the ports at which these entities connect to the network as *edge ports*. We are interested in analyzing forwarding properties between the edge ports.

A forwarding property is defined in terms of a pair of edge ports (e_s, e_d), an equivalence class (ec), and other property-specific parameters. We focus on three types of properties:
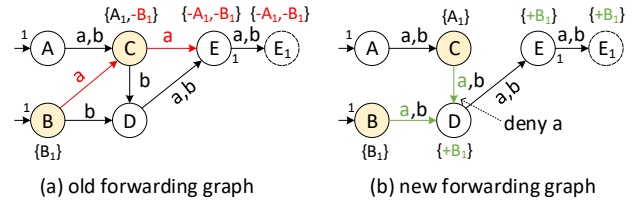

| (a) old forwarding graph | (b) new forwarding graph |

Figure 7: Differential reachability computation for the example network.

- $Reach(e_s, e_d, ec)$—packets in ec can reach e_d from e_s.
- $Waypoint(e_s, e_d, ec, w)$—packets in ec can reach e_d from e_s, traversing waypoint w.
- $LoadBalance(e_s, e_d, ec, n)$—packets in ec can reach e_d from e_s and are load balanced among n forwarding paths.

Other properties like isolation, bounded path length, etc. [4, 6], can be similarly defined. $Properties_c$ denotes the set of properties control plane c satisfies.

Given two control planes c_1 and c_2, the differences in properties are defined as $\Delta Properties_{c_1 \to c_2} := Properties_{c_2} - Properties_{c_1}$. $\Delta Properties_{c_1 \to c_2}$ is a multiset, where each item can have multiplicity $+1$ or -1. In the following, we consider the change of configuration $c_1 \to c_2$, and omit the subscripts. For example, $\Delta Properties = \{-Reach(A_1, E_1, 1.2/16)\}$, then we know this prefix is previously reachable from A_1 to E_1, but becomes unreachable after the change. This is perhaps what the operators desire if they want to prevent A_1 from reaching the prefix at E_1, or it can be a violation of operator intent if it is unexpected. As another example, $\Delta Properties = \{-Waypoint(A_1, E_1, 1.2/16, C)\}$ means this prefix will no longer traverse the waypoint C, which may violate security policies. Finally, $\Delta Properties = \{-LoadBalance(B_1, E_1, 1.2/16, 2), +LoadBalance(B_1, E_1, 1.2/16, 1)\}$ means the number of (disjoint) paths between B_1 and E_1 decreases from 2 to 1, which may cause congestion. By looking at differences in properties, instead of compliance of (all or user-specified) properties, operators can better understand the impact of prospective changes to their networks.

6.2 Computing differential properties

In this section, we show how DNA incrementally computes differential properties. We use differential reachability as an example, and discuss how to extend to the other two properties. The left of Figure 7 shows the forwarding graph of the running example. There are three edge ports, and here we only show the reachability from port 1 of A and B (denoted as A_1 and B_1), to port 1 of E (denoted as E_1).

A straightforward way to compute differential reachability is to compute the reachability for the old forwarding graph (before change) and the new forwarding graph (after change), and compute the difference. The reachability for the old graph is computed as follows. First, we start from each edge port with all ECs. Then, at each node, we com-

pute the conjunction of these ECs with the ECs marked on the edges, and move to the next hop with the conjunction of ECs. The traversal stops until another edge port is reached. For example, starting from A_1, ECs a and b can reach E_1, therefore we have a reachability $Reach(A_1, E_1, \{a, b\})$; similarly, we also have $Reach(B_1, E_1, \{a, b\})$. The reachability for the new graph is computed in a similar way, resulting in $Reach(A_1, E_1, \{b\})$ and $Reach(B_1, E_1, \{a, b\})$. Therefore $\Delta Properties_{c_1 \rightarrow c_2} = \{-Reach(A_1, E_1, a)\}$. We name this straightforward method as TraverseAll.

According to our experiment, the TraverseAll method can take tens to thousands of seconds to compute the differential rechability. To reduce the running time, we apply the following two optimizations.

(1) Only traversing with ECs whose forwarding behaviors are affected. In this example, we only need to start the traversal with EC a, since EC b is not affected. This optimization has been used by exiting realtime data plane verifiers, which check loops or blackholes by only traversing with those affected ECs. We name this method as TraverseAll-Inc. However, even there are only few ECs affected, TraverseAll-Inc still needs to enumerate all the pairs of edge ports, which can still take a long time if the network is large.

(2) Directly traversing from change points instead of from all edge ports. Here change points refer to nodes whose forwarding behaviors change (B and C in the example). Since the traversals before the change points are not affected, it is not necessary to start the traversal from each edge port, e.g., in this example we can start the traversal from B and C with EC a. To make this optimization work, we need to incrementally maintain intermediate state recording the traversal before the change points, that is, for each node, which edge ports can reach this node. For example, on the old forwarding graph, node C should know that EC a can reach C from A_1 and B_1, such that when the traversal from C reaches E_1, we can know the reachability from A_1 and B_1 to E_1.

DNA enables both these optimizations, where optimization (2) is enabled as follows. For each *node* and *ec*, DNA maintains a set $EdgeSet(ec, node)$, which stores all edge ports from which ec can reach *node*. In this example the content of $EdgeSet(a, node)$ is marked aside *node* on the forwarding graphs. When traversing, we need to update $EdgeSet$ according to the rule that if an $n_1 \in EdgeSet(ec, n_2)$, and ec can reach n_3 from n_2, then we have $n_1 \in EdgeSet(ec, n_3)$. In this example, when traversing from C to E on the old forwarding graph, since $A_1, B_1 \in EdgeSet(a, C)$, we can derive $A_1, B_1 \in EdgeSet(a, E)$. Since we are traversing the old graph, A_1, B_1 should be deleted from $EdgeSet(a, E)$, as shown on the right of Figure 7. On the contrary, when traversing the new graph, the derived entries should be inserted into $EdgeSet$. Interested reader can refer to Appendix A for the algorithm to compute differential reachability.

Computing differential waypointing and load balancing. Unlike reachability, computing differential waypointing and

load balancing requires tracking the forwarding paths between edge ports. Therefore, instead of maintaining the edge ports from which ec can reach *node*, $EdgeSet$ should maintain the forwarding path taken by ec before reaching *node*. When traversing, we need to update $EdgeSet$ according to the rule that if $p_1 \in EdgeSet(ec, n_1)$, and ec can reach n_2 from n_1, then we have $p_1||n_1 \in EdgeSet(ec, n_2)$, where p_1 is a forwarding path, and $p_1||n_1$ appends n_1 to p_1. For the running example, (B_1, B, C, E) will be deleted from $EdgeSet(a, E_1)$, and (B_1, B, D, E) will be inserted into $Edge(a, E_1)$ after the change. Suppose C is a waypoint, then the change in $EdgeSet(a, E_1)$ indicates that packets belonging to EC a, sent from B_1 will no longer traverse the waypoint C.

Computing properties under link failures. We can leverage differential property to compute properties under link failures. For example, we can compute reachability properties that hold when any single link can fail. First, we compute a set R of all reachability properties when no links fail, Then, we fail each link one by one, and after each failure we compute differential reachability. For each deletion of reachability property, we remove it from the set R. After failing each single link, R contains all reachability properties that hold under any single link failure.

7 Implementation

We implement DNA in Java. First, we use Batfish [1] to parse the configuration files into vendor-neutral configuration objects, and write a parser to generate a set of insertions of base facts for the DDlog program.

For stage 1, we model the control plane with 800 LOC in DDlog. The model currently supports BGP, OSPF, static routes, routing policies, redistribution, reflector, communities, etc. The control plane model is compiled by the DDlog compiler into a DD program for execution. For stage 2, we extend APKeep [48] to optimize the model update algorithm for batched rule updates. For stage 3, we implement an algorithm to compute differential reachability (Appendix A).

Additionally, we implement a scheduler in Python to parallelize the data plane generation. The scheduler uses the DDlog's CLI, and maintains multiple instances of the DDlog program, each of which is responsible for a group of prefixes.

8 Experiments

We evaluate DNA with both real and synthetic updates. We are interested in the following questions: (1) can DNA speed up differential property checking (§8.1 and §8.2)? (2) is incrementally simulating the control plane always better when the changes are large, and can parallelization help DNA better scale to large changes (§8.3)? (3) can DNA also speed up property checking under link failures (§8.4)?

Table 1: Types of synthesized changes.

ID	Update	Explanation
1	InterfaceUp	Bring up an interface
2	InterfaceDown	Shut down an interface
3	NetworkAdd	Add a subnet to advertise to BGP
4	NetworkDel	Delete a subnet to advertise to BGP
5	NeighborAdd	Add a BGP neighbor
6	NeighborDel	Delete a BGP neighbor
7	LocalPref	Change the local preference
8	MultiPath	Allow to select up to k paths
9	Aggregation	Add an aggregation rule
10	StaticRoute	Add a static route
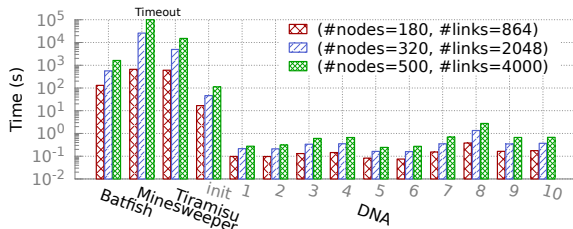
Figure 8: The time for DNA to compute differential reachability for synthetic changes on fat trees.

Setup. We run all the experiments on a server with two 12-core Intel Xeon CPUs @ 2.3GHz and 256G memory. Unless otherwise specified, a single core is used for these methods (except Batfish which is multi-threaded).

8.1 Synthetic changes

First, we evaluate the running time of DNA with synthetic changes. Specifically, we use different sizes of fat trees running BGP, where each node is assigned a distinct AS number and peers with all its adjacent nodes. We synthesize 10 different types of change, as shown in Table 1. Updates (1) and (2) can be used to simulate link failures and recovery, respectively. Updates (3) and (4) can be used to simulate changes in external routes. Update (7) adds a route map to change the local preference for routes received at one interface from 100 to 150 (more preferred).

Figure 8 reports the running time for DNA to compute differential reachability. For comparison, we also include the results for computing all-pair reachability using Batfish, Minesweeper, and Tiramisu. These tools can then compute differences of the all-pair reachability afterwards (the time to compute difference is not counted here). As we can see, DNA achieves a second-level running time for each control plane update, which is at least 3 orders of magnitude faster than existing tools—Minesweeper's [6] and Tiramisu's [4] main bottleneck is the number of links and end-host pairs, respectively. Here, init corresponds to the time for DNA to initialize, i.e., taking the original configuration snapshot as input,

and running the three stages in the same way as processing a configuration update.

Figure 9 shows a breakdown of running time for the three stages, on fat tree (#node=500, #links=4000). As shown in Figure 9(a), DNA's control plane simulation takes less than 1 second for all updates except Update 8, while Batfish, which is not incremental, always takes 24 seconds. We also compare against a version of DNA without customized functions for best route selection (§4) (DNA$^-$), and observe the simulation is ∼40% faster with our customized functions. Although the absolute savings for a large fat tree is only ∼100ms, the difference is substantial when we consider link failures: e.g., control plane simulations for all single link failures (not shown) take ∼6 minutes longer with DNA$^-$.

As shown in Figure 9(b), directly running APKeep can take as long as 0.2 seconds, while DNA can achieve running time mostly less than 0.01 seconds. In most types of changes, DNA is 10× faster than directly running APKeep.

As shown in Figure 9(c), the TraverseAll method (traversing from all edge ports with all ECs) can takes as long as 400 seconds to recompute the all reachability properties. For TraverseAll-Inc (Traverse from all edge ports with only affected ECs) method runs mostly around 1 seconds, but can take more than 10 seconds for update 7 and 8. The reason is that in these two updates, the affected ECs appear at all edge ports, and TraverseAll-Inc still needs to traverse from all edge ports. In contrast, DNA takes around 0.1 seconds, a speedup of 1-2 orders of magnitude compared to TraverseAll-Inc. For Updates 9 and 10, DNA and TraverseAll-Inc take roughly the same small amount of time. The reason is that there are a small number of affected ECs, but a large number of change points. For example, adding a static route only affects ECs overlapping with the route. However, since the route will be advertised by BGP, the ECs will change forwarding behavior at all nodes in the network. Therefore, both DNA and TraverseAll-Inc need to traverse from all edge ports.

We also experiment on fat trees running OSPF, and the trend is similar. The results can be found in Appendix §B.

8.2 Real changes

In addition to synthesized change, we also experiment with a real trace of configuration changes collected from the backbone network of a university campus. In total, the network consists of 28 routers and 50 physical links, running OSPF. The trace consists of 67 configuration snapshots spanning over three months. We compute the differences among consecutive snapshots to create 66 updates, which are fed to DNA for verification. The statistics on the network updates has already been shown in Figure 1.

Figure 10 shows the running time for the three stages of DNA. We compare the overall running time with Batfish, since Minesweeper times out (>1h per update). Also, we include the results for Baseline, which uses Batfish to generate
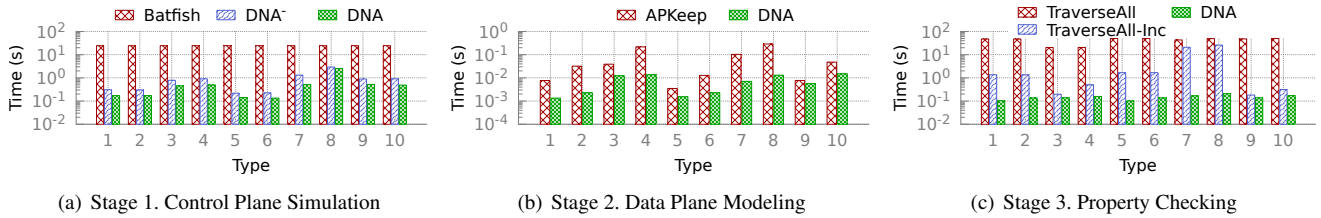
(a) Stage 1. Control Plane Simulation

(b) Stage 2. Data Plane Modeling

(c) Stage 3. Property Checking

Figure 9: The breakdown of running time for synthetic changes on fat tree (#nodes=500, #links=4000).

(a) Overall running time

(b) Control Plane Simulation
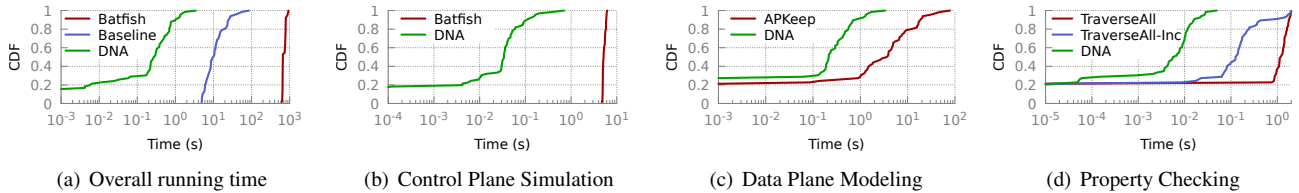
(c) Data Plane Modeling

(d) Property Checking

Figure 10: The time to verify configuration changes of campus network.

the new data plane, APKeep to update the data plane model, and Traverse-All to compute differential reachability. We can see that DNA takes <1 second per update for 90% of all updates, while Batfish takes >500 seconds per update. Baseline is faster than Batfish by using realtime verifier (i.e., APKeep), but still takes tens of seconds per update on average.

Also note that for ~20% of updates, the running time of DNA is less than 10ms. The reason is that for these updates, some interfaces or ACL rules are added without taking effect, and the output of the first stage is empty. The second and third stage are not even invoked. However, it is hard and risky to manually determine whether a change in configuration files has effect on the network, and without DNA we still need to check them using tools like Batfish or Minesweeper.

For control plane simulation, we compare the results of DNA with those of Batfish. For more than 90% of changes, the generation time is less than 0.12 seconds. While Batfish takes more than 4 seconds for each single change. This shows that incrementally generating data plane state is much faster than from scratch in real networks. For model update, DNA takes less than 1 second for 90% of changes, $10\times$ faster than APKeep. For reachability verification, DNA takes strictly less than 0.1 second, while traversing from all edge points with all or affected ECs can take several seconds.

Compared to the synthesized changes on fat trees, where stage 1 dominates the overall running time, here stage 2 dominates the overall running time. The reason is that the largest fat tree (500 nodes, running BGP) has only 5K ECs, while the campus network has 45K ECs, due to the existence of ACLs.

8.3 Large changes and parallel simulation

In the previous two experiments, we mainly focus on small configuration changes. However, a network can occasionally experience large-scale changes [27], which may affect a large number of devices. We simulate large changes by shutting down a large number of interfaces in the campus network.

Figure 11 reports the time for DNA to incrementally simulate the control plane when failing a different number of links. For comparison, we also include the results of Batfish. As we can see, when failing a small number of links (say <10), incrementally simulating the control plane is much faster; while when the number increases to over 25 (single core), the incremental simulation becomes even slower than generation from scratch. This means that incremental simulation outperforms from-scratch simulation as long as the change sizes are smaller than some threshold (in our case, 50%). While since most of real changes are small (§2.1), incremental simulation would mostly be a better choice.

We also note that parallelizing the control plane simulation can increase such a threshold. Specifically, incremental simulation with 24 cores has larger improvement for larger changes. Even all links in the network were disconnected, the incremental simulation time is still comparable with from-scratch generation using Batfish.

We further study the effect of parallelizing control plane simulation on different size fat trees. For both BGP and OSPF, we randomly fail one node as well as all links connected to this node. Figure 12 reports the control plane simulation time for DNA with different number of cores. We can see the simulation speed increases with the number of cores. Due to the overhead of parallel simulation, the speed-up is not significant when the total time is already small (e.g., <1 second).

8.4 Enumerating link failures

One verification task is to check reachability under any single link failure. Using Batfish we need to enumerate each link failure and Minesweeper relies on SMT solvers to search for a counterexample where a link failure breaks reachability. DNA can leverage the similarity between the no link failure and single link failure to enumerate all link failures efficiently.

In this experiment, we evaluate the time to check reachability policies with any single link failure, i.e., whether two
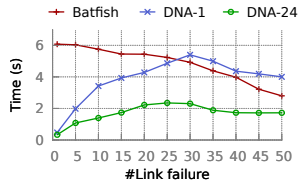
Figure 11: The running time for control plane simulation on the campus network.
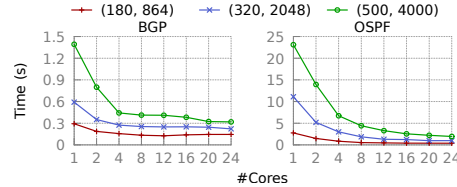

Figure 12: The running time for control plane simulation on fat trees, for one node failure with multiple cores.
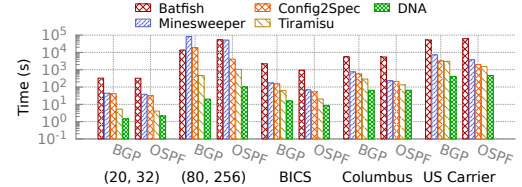

Figure 13: The total time for checking reachability under any single link failure. (n_1, n_2) represents fat tree with n_1 nodes and n_2 links.

hosts or ports are always reachable if any link can fail. We use two sizes of fat trees with 20 and 80 nodes, and three ISP topologies from Config2Spec. For each topology, we fail a link each time, and let DNA compute differential reachability.

For comparison, we run Batfish, Minesweeper, and Tiramisu to check the reachability between each pair of hosts on the same networks. We also include Config2Spec for comparison, since it is shown to outperform both Batfish and Minesweeper when checking all-pair reachability under link failures. Since Config2Spec also checks other properties like load balancing, we modify it to let it only compute all-pair reachability for a fair comparison.

As shown in Figure 13, DNA is at least 10× faster than Batfish and Minesweeper, 3× faster than Config2Spec, on all topologies. DNA is also faster than Tiramisu by 10× on the 80-node fat tree. The speedup of DNA on the three ISP topologies is not as remarkable as on fat trees. The reason is that the links on the ISP topologies are not as redundant as on fat trees, and a single link failure has a larger impact on reachability. The above results imply that by leveraging the similarity among network snapshots with and without a failed link, DNA can check reachability policies under any single link failure faster than existing control plane verifiers.

9 Related Work

Control plane simulation/emulation. Control plane verifiers have employed several approaches for simulating the control plane including: a Datalog engine [14], an explicit state model checker [37], a generalized variant of Dijkstra's algorithm [34], abstract interpretation [7], and custom simulation engines [1, 38, 45]. However, all these approaches restart the simulation from scratch when the control plane changes, and do not reuse any of the state from prior simulations. The preliminary version of this paper [47] introduces incremental network configuration verification, but does not parallelize incremental control plane simulation, and has limited support for incremental data plane modeling and property checking. Control plane emulators [32] can accommodate control plane changes in an incremental manner, but they scale poorly due to their use of actual routing protocol implementations.

Symbolic control plane verifiers. Symbolic control plane verifiers characterize the space of data planes the control

plane may produce using graph algorithms [4, 15], SMT constraints [6, 43], or binary decision diagrams [13]. Even though some incremental graph algorithms exist [29] and SMT solvers offer some support for incremental solving [11], these capabilities are not sufficient to accommodate arbitrary control plane changes.

Data plane verifiers. Realtime data plane verifiers [18, 23, 26, 44, 48] can quickly analyze data plane changes (e.g., forwarding rule insertions), but cannot directly analyze configuration changes. DNA uses a realtime data plane verifier, APKeep [48], as one of its building blocks, and modifies AP-Keep to batch data plane changes for efficient processing, Other realtime data plane verifiers could also be modified to batch data plane changes and be used in DNA. NoD [33] uses Datalog, but NoD is not realtime.

Specification mining. Policy Units [8], Config2Spec [9], and Anime [25] infer a network's end-to-end behaviors from its configurations. We could compute differences in end-to-end behavior by applying such tools before and after a configuration change. However, due to the prevalence of small configuration changes (§2.1), such an approach unnecessarily duplicates computation in the same manner as applying a control plane verifier before and after changes (§1).

10 Conclusion

Differential Network Analysis (DNA) addresses a critical gap in control plane analysis: efficiently and effectively identifying differences in end-to-end forwarding behaviors arising from control plane changes. DNA uses a three-stage process that leverages advances in differential dataflow programming frameworks and data plane verifiers, along with domain-specific optimizations. Our evaluations using real and synthetic control plane changes show that DNA is able to compute differences in reachability in a few seconds—up to 3 orders of magnitude faster than state-of-the-art control plane verifiers. Thus, DNA provides a promising approach for operators to assess the impact of control plane changes.

References

[1] Batfish. https://github.com/batfish/batfish.

[2] Differential Datalog (DDlog). https://github.com/vmware/differential-datalog.

[3] Internet2 - visible backbone. https://vn.net.internet2.edu/Internet2/.

[4] A. Abhashkumar, A. Gember-Jacobson, and A. Akella. Tiramisu: Fast and general network verification. In *USENIX NSDI*, 2020.

[5] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn. Design and implementation of the LogicBlox system. In *ACM SIGMOD*, 2015.

[6] R. Beckett, A. Gupta, R. Mahajan, and D. Walker. A general approach to network configuration verification. In *ACM SIGCOMM*, 2017.

[7] R. Beckett, A. Gupta, R. Mahajan, and D. Walker. Abstract interpretation of distributed network control planes. In *ACM POPL*, 2020.

[8] T. Benson, A. Akella, and D. A. Maltz. Mining policies from enterprise network configuration. In *ACM IMC*, 2009.

[9] R. Birkner, D. Drachsler-Cohen, L. Vanbever, and M. Vechev. Config2Spec: Mining network specifications from network configurations. In *USENIX NSDI*, 2020.

[10] G. Comarela, G. Gürsun, and M. Crovella. Studying interdomain routing over long timescales. In *ACM IMC*, 2013.

[11] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[12] A. Elmokashfi, A. Kvalbein, and C. Dovrolis. BGP churn evolution: A perspective from the core. *IEEE/ACM Transactions on Networking*, 20(2):571–584, 2012.

[13] S. K. Fayaz, T. Sharma, A. Fogel, R. Mahajan, T. Millstein, V. Sekar, and G. Varghese. Efficient network reachability analysis using a succinct control plane representation. In *USENIX OSDI*, 2016.

[14] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. A general approach to network configuration analysis. In *USENIX NSDI*, 2015.

[15] A. Gember-Jacobson, R. Viswanathan, A. Akella, and R. Mahajan. Fast control plane analysis using an abstract representation. In *ACM SIGCOMM*, 2016.

[16] A. Gember-Jacobson, W. Wu, X. Li, A. Akella, and R. Mahajan. Management plane analytics. In *ACM IMC*, 2015.

[17] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *ACM SIGCOMM*, 2011.

[18] A. Horn, A. Kheradmand, and M. R. Prasad. Delta-net: Real-time network verification using atoms. In *USENIX NSDI*, 2017.

[19] K. Jayaraman, N. Bjørner, J. Padhye, A. Agrawal, A. Bhargava, P. C. Bissonnette, S. Foster, A. Helwer, M. Kasten, I. Lee, A. Namdhari, H. Niaz, A. Parkhi, H. Pinnamraju, A. Power, N. M. Raje, and P. Sharma. Validating datacenters at scale. In *ACM SIGCOMM*, 2019.

[20] W. M. Johnston, J. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36(1):1–34, 2004.

[21] H. Jordan, B. Scholz, and P. Subotić. Soufflé: On synthesis of program analyzers. In *International Conference on Computer Aided Verification*, 2016.

[22] S. K. R. Kakarla, A. Tang, R. Beckett, K. Jayaraman, T. D. Millstein, Y. Tamir, and G. Varghese. Finding network misconfigurations by automatic template inference. In *USENIX NSDI*, 2020.

[23] P. Kazemian, M. Chan, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *USENIX NSDI*, 2013.

[24] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *USENIX NSDI*, 2012.

[25] A. Kheradmand. Automatic inference of high-level network intents by mining forwarding patterns. In *ACM Symposium on SDN Research*, 2020.

[26] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey. VeriFlow: Verifying network-wide invariants in real time. In *USENIX NSDI*, 2013.

[27] H. Kim, T. Benson, A. Akella, and N. Feamster. The evolution of network configuration: a tale of two campuses. In *ACM IMC*, 2011.

[28] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark. Kinetic: Verifiable dynamic network control. In *USENIX NSDI*, 2015.

[29] Y. Li, J. Jia, X. Hu, and J. Li. Real time control plane verification. In *Proceedings of the ACM SIGCOMM 2019 Workshop on Networking and Programming Languages*, pages 2–2, 2019.

[30] B. Liu, A. Kheradmand, M. Caesar, and P. B. Godfrey. Towards verified self-driving infrastructure. In *ACM HotNets*, 2020.

[31] H. H. Liu, X. Wu, W. Zhou, W. Chen, T. Wang, H. Xu, L. Zhou, Q. Ma, and M. Zhang. Automatic life cycle management of network configurations. In *SIGCOMM Workshop on Self-Driving Networks*, 2018.

[32] H. H. Liu, Y. Zhu, J. Padhye, J. Cao, S. Tallapragada, N. P. Lopes, A. Rybalchenko, G. Lu, and L. Yuan. Crystalnet: Faithfully emulating large production networks. In *ACM SOSP*, 2017.

[33] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese. Checking beliefs in dynamic networks. In *USENIX NSDI*, 2015.

[34] N. P. Lopes and A. Rybalchenko. Fast BGP simulation of large datacenters. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, 2019.

[35] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard. Differential dataflow. In *CIDR*, 2013.

[36] D. Plonka and A. J. Tack. An analysis of network configuration artifacts. In *Proceedings of the 23rd Large Installation System Administration Conference*, 2009.

[37] S. Prabhu, K.-Y. Chou, A. Kheradmand, P. Godfrey, and M. Caesar. Plankton: Scalable network configuration verification through model checking. In *USENIX NSDI*, 2020.

[38] B. Quoitin and S. Uhlig. Modeling the routing of an autonomous system with C-BGP. *IEEE Network*, 19(6):12–19, 2005.

[39] L. Ryzhyk and M. Budiu. Differential Datalog. In *International Workshop on the Resurgence of Datalog in Academia and Industry*, 2019.

[40] S. Steffen, T. Gehr, P. Tsankov, L. Vanbever, and M. Vechev. Probabilistic verification of network configurations. In *ACM SIGCOMM*, 2020.

[41] Y. E. Sung, X. Tie, S. H. Y. Wong, and H. Zeng. Robotron: Top-down network management at facebook scale. In *ACM SIGCOMM*, 2016.

[42] D. Turner, K. Levchenko, A. C. Snoeren, and S. Savage. California fault lines: understanding the causes and impact of network failures. In *ACM SIGCOMM*, 2010.

[43] K. Weitz, D. Woos, E. Torlak, M. D. Ernst, A. Krishnamurthy, and Z. Tatlock. Scalable verification of border gateway protocol configurations with an SMT solver. In *ACM OOPSLA*, 2016.

[44] H. Yang and S. S. Lam. Real-time verification of network properties using atomic predicates. In *IEEE ICNP*, 2013.

[45] F. Ye, D. Yu, E. Zhai, H. H. Liu, B. Tian, Q. Ye, C. Wang, X. Wu, T. Guo, C. Jin, et al. Accuracy, scalability, coverage: A practical configuration verifier on a global wan. In *ACM SIGCOMM*, 2020.

[46] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic test packet generation. In *ACM CoNEXT*, 2012.

[47] P. Zhang, Y. Huang, A. Gember-Jacobson, W. Shi, X. Liu, H. Yang, and Z. Zuo. Incremental network configuration verification. In *ACM HotNets*, 2020.

[48] P. Zhang, X. Liu, H. Yang, N. Kang, Z. Gu, and H. Li. APKeep: Realtime verification for real networks. In *USENIX NSDI*, 2020.

A   An Algorithm for Computing Differential Reachability

Algorithm 1 summarizes how DNA traverses the new forwarding graph while avoiding redundant traversal. The traversal for the old forwarding graph is the same except Lines 6, 7, and 10. For ease of notation, we represent each edge port as a special node termed *edge node*. We traverse from each change point *loc* with all the affected ECs *pkts* (Lines 2-3). Suppose we are traversing from v to w, then we generate $pkts'$ by intersecting *pkts* with the ECs that can be forwarded to w according to the data plane model (Line 15), and record this information in R (Line 16). $(loc, w, pkts) \in R$ if *pkts* can reach w from the change point *loc*. If w is another change point, where the affected ECs are $pkts''$, the traversal continues to w with $pkts' \backslash pkts''$ (Line 17-18). In this sense, we *delegate* the traversal of common ECs $pkts' \cap pkts''$ to the traversal starting from w, therefore avoiding the redundant traversal of common ECs. If w is not a change point, the traversal continues to w with $pkts'$ (Line 19-20). The traversal ends when the set of ECs becomes empty (Line 12-13).

After all traversals finish, the algorithm iterates over all affected EC δ, and for each EC, extracts the forwarding paths of δ (Line 5). Then, for each link (loc, w) on the path, it updates $EdgeSet(\delta, w)$ according to topological order of w (Line 6-7). If w is an edge node, it updates the reachability matrix *Reach* (Line 8-10).

Algorithm 1: `DiffReach`(*Graph*, *Changes*)

> **Input:** *Graph*: the forwarding graph; *Changes*: the set of
> data plane model changes.
> **Output:** *Diff*: the differential reachability.

1 $R \leftarrow \{\}$;
2 **foreach** $(loc, pkts) \in Changes$ **do**
3   Traverse$(loc, loc, pkts)$;

4 **foreach** $\delta \in \bigcup_{(loc, pkts) \in Changes} pkts$ **do**
5      $Path(\delta) \leftarrow \{(loc, w) | (loc, w, pkts) \in R, \delta \in pkts\}$;
6   **foreach** $(loc, w) \in Path(\delta)$ **do**
7          $EdgeSet(\delta, w) \leftarrow EdgeSet(\delta, w) \cup EdgeSet(\delta, loc)$;
8   **if** *w is an edge node* **then**
9   **foreach** $e \in EdgeSet(\delta, loc)$ **do**
10                  $Diff \leftarrow Diff \cup \{+Reach(e, w, \delta)\}$;

11 **Function** Traverse$(loc, v, pkts)$:
12   **if** $pkts = \emptyset$ **then**
13   return;
14   **foreach** $(v, w) \in Graph$ **do**
15          $pkts' \leftarrow pkts \cap EC(v, w)$;
16          $R \leftarrow R \cup \{(loc, w, pkts')\}$;
17   **if** $(w, pkts'') \in Changes$ **then**
18   Traverse$(loc, w, pkts' \setminus pkts'')$;
19   **else**
20   Traverse$(loc, w, pkts')$;

C Customized function for BGP best route selection in DDlog-based model

The following shows the code snippet of the function for best routes selection (simplified for ease of presentation).

```
BestRoute(route.node, route.dst, route.nexthop, ...) :-
  MatchedRIBIn[route1],
  var node = route1.node,
  var dst = route1.dst,
  var route = Aggregate((node, dst), select_best(route1))
    .

function select_best(g: Group<'K, AdjRIBIn>): AdjRIBIn {
  var route = group_first(g);
  for (route1 in g) {
    if (route1.LocalPref > route.LocalPref) route = route1
    else if (route1.LocalPref == route.LocalPref) {
      if (len(route1.path) < len(route.path)) route =
        route1
    // origin, MED, eBGP < iBGP, router-id, ...
  }};
  route // return the best route
}
```

Table 2: Types of synthesized changes (OSPF). The IDs continue after Table 1.

ID	Update	Explanation
11	InterfaceUp	Bring up an interface
12	InterfaceDown	Shutdown an interface
13	LinkCost	Change the cost of one link
14	MultiPath	Allow to select up to k paths
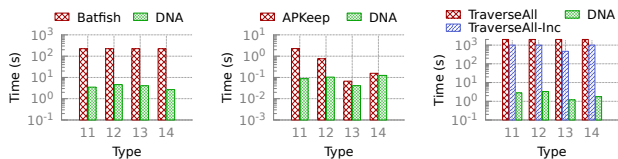
(a) Stage 1. Control Plane Simulation   (b) Stage 2. Data Plane Modeling   (c) Stage 3. Property Checking

Figure 14: The breakdown of running time for synthetic changes on fat tree (#nodes=500, #links=4000).

B Experiments for Fat Tree Running OSPF

Figure 14 shows the breakdown of running time for DNA, on fat tree running OSPF. The updates are described in Table 2.

KATRA: Realtime Verification for Multilayer Networks

Ryan Beckett
Microsoft

Aarti Gupta
Princeton University

Abstract

We present a new verification algorithm to efficiently and incrementally verify arbitrarily *layered* network data planes that are implemented using packet encapsulation. Inspired by work on model checking of pushdown systems for recursive programs, we develop a verification algorithm for networks with packets consisting of *stacks* of headers. Our algorithm is based on a new technique that lazily "repairs" a decomposed stack of header sets on demand to account for cross-layer dependencies. We demonstrate how to integrate our approach with existing fast incremental data plane verifiers and have implemented our ideas in a tool called KATRA. Evaluating KATRA against an alternative approach based on equipping existing incremental verifiers to emulate finite header stacks, we show that KATRA is between 5x-32x faster for packets with just 2 headers (layers), and that its performance advantage grows with both network size and layering.

1 Introduction

The success of networks is in part due to their *layered* design where different protocol layers are delegated different responsibilities. For instance, many networks, including virtual networks [3, 16, 31], are designed in an overlay/underlay pattern that is implemented by encapsulating packets, e.g. using IP-in-IP [34], IP GRE [18], or VXLAN [30] tunnels. In a wide-area network (WAN), routing protocols such as iBGP and SDN solutions such as SWAN [19] rely on an underlying label-switching protocol like MPLS. In routers, Ethernet frames are encapsulated in IP headers to implement forwarding, and new and emerging technologies such as SD-WAN can connect two WANs together by tunneling packets to each other securely using IPSEC [28].

However, while this layered design has proven successful, allowing each layer to hide many details from others, it also makes operating networks reliably a challenge as many bugs can sit in the intersection of one or more of these layers [37]. Given the pervasiveness of layering as a fundamental design pattern in networks, it is critical that we be able to ensure the reliability of networks using this design. In practice, the implementation of layering is often extraordinarily complex. For instance, packets going over AT&T's backbone network consist of as many as eleven encapsulated headers [44].

To ensure the safe operation of multilayer networks, a natural technology to employ is that of network verification, which has emerged as a viable technique to proactively catch bugs and misconfigurations related to automation and human error. Employed by nearly all major cloud providers now [5, 17, 21, 39, 45], network verification has witnessed substantial practical use in industry as researchers have iteratively improved upon the scalability, responsiveness, and expressiveness of the underlying verification tools and techniques. Many of these tools are engineering marvels – through complex data structures and algorithms, they enable efficient verification of new network changes in milliseconds.

While recent progress in network verification has been substantial, existing tools typically analyze a single layer or component of the network stack. For example, most verification tools in use today analyze only the simple IP-based forwarding tables and ACLs [4, 7, 20, 21, 26, 27, 29, 41, 42, 46] governed by the control plane. To verify a network with N layers using verification tools today, one possibility is to model packets in the network as consisting of N duplicate copies of different headers (e.g., an IPv4 header). While this approach is possible, and researchers have proposed this approach in some prior work [42, 46], it suffers from two major problems.

The first problem is that the number of layers N may not be known a priori by the user of the verification tool. For instance, a network making use of the MPLS fast reroute (FRR) [14, 25, 33] protects links against failure by rerouting traffic that would have gone over the failed link along a predetermined backup tunnel. FRR schemes may encapsulate a packet's header a number of times that depends on the number of failures encountered by the packet along its path. Moreover, if a user provides an incorrect (low) estimate of N, then the verifier may report both false positives and false negatives.

The second problem is that, even when the user is able to statically determine N, modeling this many packet headers simultaneously leads to a highly inefficient representation. Most network verifiers work by modeling the sets of packets that are reachable from each ingress and egress point in the network [27, 41, 42, 46], and such sets may grow substantially larger and more complex to process when capturing the dependencies between different headers in the packet. In our experiments (§7), even with just two layers, this simple approach can be anywhere from 5x-32x slower than necessary.
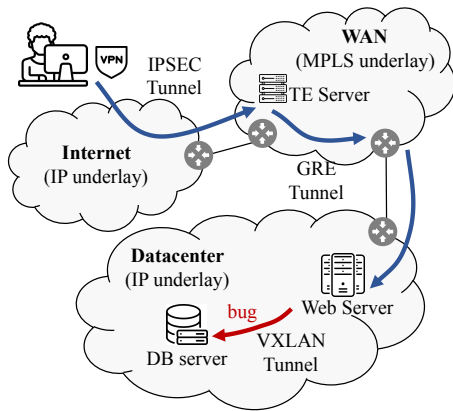
Figure 1: A user attempts to access a corporate web service.

Networking example	Layering mechanism
campus network isolation [44]	VLAN, VXLAN [30]
virtual private network [15]	IP-IP [34], IP GRE [18]
3G/4G mobile packet core [38]	GPRS tunneling [1]
interior gateway protocol [32]	LSPs, MPLS [35]
performance proxy [8]	IPSEC [28]
traffic engineering [43]	IP GRE [18]
software-defined WAN [13]	IPSEC [28]

Table 1: Example network services features with layering.

To address these problems, in this paper, we present KA-TRA, the first tool for real-time verification of layered networks – those networks that manipulate *stacks* of headers through packet encapsulation. Inspired by recent work on analyzing MPLS [22, 23], we leverage ideas from the verification of pushdown systems [36] used to model recursive programs. A key new idea is to keep a decomposed representation of the set of header stacks as a stack of header sets where the set of headers at one layer in the stack are treated as independent of those in another. When an operation modifies and later restricts packets at a given layer, it may be necessary to go back and "repair" the sets of packets representative of previous headers in the stack to account for this change.

We demonstrate how we can integrate our ideas with existing incremental data plane verifiers, and we implement KATRA as an extension to the state-of-the-art verifier AP-Keep [46]. In addition to being able to reason about arbitrarily large header stacks, for networks consisting of just two layers, we further demonstrate that KATRA can verify properties 5x-32x faster than an approach based on extending existing algorithms with a finite header stack encoding.

Our contributions with KATRA are the following:

- We present a new formal model, and its semantics, for layered networks that supports arbitrarily nested packet encapsulations and decapsulations.

- We develop an efficient algorithm for verifying layered networks in our model. The algorithm is based on a new notion of *partial* equivalence class as well as a new decomposed representation for symbolic header stacks.

- We implement our ideas in a tool called KATRA, which integrates our new model and algorithm with the state-of-the-art incremental verification approach based on APKeep [46].

- We evaluate KATRA against a baseline based on a finite header stack representation and demonstrate that KATRA is 5x-32x faster with this speedup growing larger with both network size and the number of network layers.

2 Motivation and Background

Protocol *layering* in networks is used pervasively as a way to separate concerns and build new features and services atop existing infrastructure. Consider the scenario depicted in Figure 1. In the example, an employee of a company attempts to access a corporate web service, that is hosted in the cloud, from their home. The employee uses a secure VPN connection so their traffic is encapsulated in an IPSEC [28] tunnel before being forwarded over the Internet. When the traffic traverses the cloud provider's wide area network (WAN), a traffic engineering server selects an egress point for a nearby data center. The WAN forwards packets to this egress using an IP-GRE tunnel over its MPLS-based core. Once the traffic reaches the data center, it is forwarded to the requested web server. The web server must now access data from a database (DB) server configured in the same virtual overlay network. The web server thus sends traffic to the DB server using a VXLAN tunnel configured atop an IP-based datacenter fabric.

In the example, layer 2 and layer 3 forwarding elements are combined to implement several abstractions. While this approach to protocol layering is powerful, if any one of the forwarding policies at any layer in the example is misconfigured, then the user will not be able to reach the intended service. For instance, if there were a misconfigured security rule in the overlay network between the web and DB servers then the user may lose connectivity to the web service. Similarly, a misconfiguration in the datacenter's IP-based underlay network could break connectivity. To make matters worse, when network forwarding bugs span multiple layers, identifying the root cause of the issue can be complicated as it may require coordination between multiple different teams spread across one or more organizations [37], each with only a partial view of the network as a whole.

Multilayer verification. Network verification is a natural fit to ensure the correctness of multilayer networks, yet verifiers today were not built with layering in mind. Existing verifiers assume that packets have a fixed size header rather than an expandable *stack* of headers. While it is sometimes possible to retroactively analyze such multilayer networks using existing verifiers by pessimistically modeling a "worst case" fixed size header stack, doing so is often highly inefficient.

3   Layered Network Model

Rather than model networks with fixed size headers, as is done by existing network verifiers, in this section we define a new network model that includes protocol layering as a first class concern. We model multilayer networks as operating on an unbounded *stack* of headers. We then demonstrate that one can view both single-layer and multi-layer networks as instances of our general model.

3.1   Notation and Preliminaries

Before defining our network model, we first introduce some notation and preliminary definitions.

Definition 3.1 (Sequences). *For a set X, we use the notation X^* to mean the set of all possible sequences of elements of X. We define a sequence $\sigma \in X^*$ inductively as either ε representing the empty sequence, or the concatenation $(\sigma' \cdot x)$ of another sequence $\sigma' \in X^*$ together with an element $x \in X$.*

For simplicity of notation, we sometimes write a sequence $\sigma \in X^*$ as $\sigma = x_0 \cdot x_1 \cdot \cdots \cdot x_n$ where $x_i \in X$ and x_0 is the first element of the sequence and x_n is the last element, and we omit writing out the ε. Concatenation of two sequences is defined recursively as $\sigma \cdot \varepsilon = \sigma$ and $\sigma \cdot (\sigma' \cdot x) = (\sigma \cdot \sigma') \cdot x$ As a shorthand, for a sequence $\sigma = \sigma' \cdot x$ we define $\text{top}(\sigma) = x$ and $\text{bot}(\sigma) = \sigma'$. These two partial functions are undefined when $\sigma = \varepsilon$. Finally, we write $|\sigma|$ to mean the length of a sequence such that $|\varepsilon| = 0$ and $|\sigma \cdot x| = 1 + |\sigma|$. A stack is a sequence σ where the top of the stack is given by $\text{top}(\sigma)$.

For sets X and Y, we use the standard notation Y^X or $X \to Y$ to mean the set of functions from X to Y. Similarly, we use the notation $X \hookrightarrow Y$ to represent partial functions from X to Y. Given a (potentially partial) function f from X to Y and function g from Y to Z, we write $f \circ g$ to define their composition, from X to Z.

3.2   Formal network model

We define a network $\mathcal{N}$ as a tuple $\langle V, E, \mathcal{H}, \mathcal{T}, \mathcal{R} \rangle$ where:

- V is a set of vertices, and $E \subseteq 2^{V \times V}$ is a set of edges. Edges are unidirectional and we represent bidirectional edges with a pair of edges. For edge $e = \langle u, v \rangle$, we use the notation $\text{src}(e) = u$ and $\text{tgt}(e) = v$.

- $\mathcal{H}$ is a set of valid headers for the protocols in use.

- $\mathcal{T}$ is a set of transformations, which are partial functions over packet headers of type $\mathcal{T} \subseteq 2^{\mathcal{H} \hookrightarrow \mathcal{H}}$.

- $\mathcal{R}$ is a set of rules $\mathcal{R} \subseteq \mathbb{N} \times E \times 2^{\mathcal{H}} \times \mathcal{T}$. For rule $r \in \mathcal{R}$ where $r = \langle p, e, m, \tau \rangle$. We use the notation $\text{priority}(r) = p, \text{edge}(r) = e, \text{match}(r) = m$, and $\text{modify}(r) = \tau$ to refer to the components of the rule.

Intuitively, lower priority rules take precedence over those with higher priority at a given node. The best rule for a given header at a node is the rule with lowest priority at that node that also matches the header.

Definition 3.2 (Best rule). *For a node $u \in V$ and a header $h \in \mathcal{H}$, we define the best matching rule at u as:* $\Omega(u,h) = \min_{\text{priority}}\{r \in \mathcal{R} \mid \text{src}(\text{edge}(r)) = u, \ h \in \text{match}(r)\}$

For simplicity, we assume that there is always a unique best rule for a given header. This is ensured by requiring that (1) rule priorities must be unique, and (2) all headers are matched by at least one rule. In practice, one can ensure this requirement is met by adding a maximum priority default rule that matches all other unmatched headers.

3.3   Network semantics

Given a header $h \in \mathcal{H}$ and an initial node $u \in V$, which we call a located packet $L = V \times \mathcal{H}$, the network produces a sequence of new located packets to capture the packet's history as it goes through the network. Specifically, we define the semantics of a network $\mathcal{N}$ as a function $[\![\mathcal{N}]\!]_i : L \to L^*$ that takes an initial located packet to a trace of located packets through the network for a given number of steps $i \in \mathbb{N}$:

$$[\![\mathcal{N}]\!]_i\langle u, h \rangle = \begin{cases} \varepsilon \cdot \langle u, h \rangle & \text{if } i = 0 \\ \sigma & \text{elif } \tau(h') \text{ undefined} \\ \sigma \cdot \langle v, \tau(h') \rangle & \text{otherwise} \end{cases}$$

where $\sigma = [\![\mathcal{N}]\!]_{i-1}\langle u, h \rangle$ and $\text{top}(\sigma) = \langle u', h' \rangle$ and:

$$\begin{aligned} \tau &= \text{modify}(\Omega(u', h')) \\ v &= \text{tgt}(\text{edge}(\Omega(u', h'))) \end{aligned}$$

Definition 3.3 (Packet termination). *We say network $\mathcal{N}$ has terminated a located packet ℓ after i steps, written $\mathcal{N} \otimes \langle i, \ell \rangle$, if the trace no longer changes:* $[\![\mathcal{N}]\!]_i \ell = [\![\mathcal{N}]\!]_{i-1} \ell$.

3.4   Lifting networks to layered networks

To implement layering, conceptually packets contain an unbounded stack of headers to which the network can push or pop. A layered network is just an instance of a network that processes *stacks* of headers:

Definition 3.4 (Multilayer network). *A multilayer network is an instance of a network $\mathcal{N} = \langle V, E, \mathcal{H}^*, \mathcal{T}, \mathcal{R} \rangle$ over sequences of headers $\mathcal{H}^*$ with some restrictions on $\mathcal{T}$ and $\mathcal{R}$.*

Primarily, we require that every transformation $\tau \in \mathcal{T}$ and every rule match set $\text{match}(r)$ for $r \in \mathcal{R}$ only inspects or modifies the top of the stack[1]. In other words, we often write that $\tau(\sigma \cdot x) = \sigma \cdot \tau(x)$. And that $\sigma \cdot h \in \text{match}(r) \iff h \in \text{match}(r)$ as though τ and $\text{match}(r)$ were defined over $\mathcal{H}$. The

---

[1]This restriction can express encapsulation and decapsulation in real networks yet also makes verification tractable.
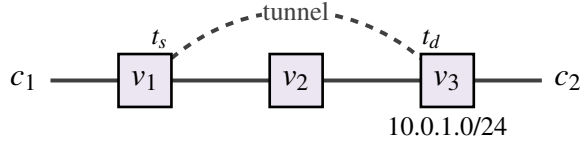
rule	p	e	m	τ
r_0	100	$\langle c_1, v_1 \rangle$	$\mathcal{H}^*$	τ_{id}
r_1	100	$\langle v_1, v_1 \rangle$	$\phi_{\text{dst}}(23.1.4.0/24)$	$\tau_{\text{push}} \circ \tau_{\text{tunl}}$
r_2	200	$\langle v_1, v_2 \rangle$	$\phi_{\text{dst}}(10.0.1.0/24)$	τ_{id}
r_3	300	$\langle v_1, v_1 \rangle$	$\mathcal{H}^*$	τ_{drop}
r_4	100	$\langle v_2, v_3 \rangle$	$\phi_{\text{dst}}(10.0.1.0/24)$	τ_{id}
r_5	200	$\langle v_2, v_2 \rangle$	$\mathcal{H}^*$	τ_{drop}
r_6	100	$\langle v_3, v_3 \rangle$	$\phi_{\text{src}}(t_s) \cap \phi_{\text{dst}}(t_d)$	τ_{pop}
r_7	200	$\langle v_3, c_2 \rangle$	$\mathcal{H}^*$	τ_{id}
r_8	100	$\langle c_2, c_2 \rangle$	$\mathcal{H}^*$	τ_{delv}

Figure 2: Example formulation of a network with a single tunnel between v_1 and v_3. We use the notation $\phi_f(P)$ for set P as a shorthand to mean packets where the field f is contained in P. Thus $\phi_{\text{dst}}(P) = \{\sigma \in \mathcal{H}^* \mid \langle d, s \rangle \in \text{top}(\sigma), d \in P\}$. Tunneled packets are encapsulated by first executing τ_{push} to duplicate the top-most header before modifying this header copy with τ_{tunl} to set to source ip to $t_s = 10.0.2.0$ and the destination ip to $t_d = 10.0.1.0$. Packets are then forwarded according to the underlay network towards $t_d \in 10.0.1.0/24$ hosted at v_3. Packets at v_3 are decapsulated by popping the top-most header and then delivered to client c_2.

only exception to this restriction is for two special transformations τ_{push}, which makes a new copy of the current top of the stack, and τ_{pop}, which drops or decapsulates the top of the stack. More specifically, we define $\tau_{\text{push}}(\sigma \cdot x) = \sigma \cdot x \cdot x$ and we define $\tau_{\text{pop}}(\sigma \cdot x \cdot y) = \sigma \cdot x$. Both transformations are undefined otherwise and may be composed with other transformations (e.g., $\tau \circ \tau_{\text{push}}$).

Example Network. To make the model more concrete, we show an example of its instantiation in Figure 2. The network in the figure is given by the tuple $\mathcal{N} = \langle V, E, \mathcal{H}^*, \mathcal{T}, \mathcal{R} \rangle$ where the nodes and edges are defined by the sets:

$$V = \{c_1, v_1, v_2, v_3, c_2\}$$
$$E = \{\langle c_1, v_1 \rangle, \langle v_1, v_1 \rangle, \langle v_1, v_2 \rangle, \langle v_2, v_3 \rangle, \langle v_3, v_3 \rangle, \langle v_3, c_2 \rangle\}$$

Note that the edges include self edges (e.g., $\langle v_1, v_1 \rangle$) to model recursive lookup for forwarding and tunneling.

The set of headers $\mathcal{H} = \{\langle d, s \rangle \mid d, s \in \{0, \dots, 2^{32} - 1\}\}$ defines headers consisting of two 32-bit IP address fields for destination and source IP, and $\mathcal{H}^*$ is all sequences (stacks) of such headers. The set of transformations for the network is given by $\mathcal{T} = \{\tau_{\text{id}}, \tau_{\text{drop}}, \tau_{\text{delv}}, \tau_{\text{push}} \circ \tau_{\text{tunl}}, \tau_{\text{pop}}\}$. The transformation τ_{id} is the identity transformation such that $\tau_{\text{id}}(\sigma) = \sigma$, τ_{drop} and τ_{delv} are transformations that are undefined for all inputs and thus terminate traffic, τ_{tunl} is a transformation that

i	$\text{top}(\llbracket \mathcal{N} \rrbracket_i \langle c_1, \langle d, s \rangle \rangle)$	description
0	$\langle c_1, \langle d, s \rangle \rangle$	*forward to v_1*
1	$\langle v_1, \langle d, s \rangle \rangle$	*encapsulate*
2	$\langle v_1, \langle d, s \rangle \cdot \langle t_s, t_d \rangle \rangle$	*forward to v_2*
3	$\langle v_2, \langle d, s \rangle \cdot \langle t_s, t_d \rangle \rangle$	*forward to v_3*
4	$\langle v_3, \langle d, s \rangle \cdot \langle t_s, t_d \rangle \rangle$	*decapsulate*
5	$\langle v_3, \langle d, s \rangle \rangle$	*forward to c_2*
≥ 6	$\langle c_2, \langle d, s \rangle \rangle$	*delivered*

Table 2: Trace of a packet with source s and destination d from client node c_1 in the network shown in Figure 2.

rewrites the source IP address to t_s and the destination IP address to t_d. The composed transformation $\tau_{\text{push}} \circ \tau_{\text{tunl}}$ first creates a copy of the top of the stack and then rewrites the IP addresses according to τ_{tunl} for the encapsulated header.

A trace through the network given by the semantics (see §3.3) represents the packet forwarding in the network. Consider sending an initial packet from client c_1 with some destination address $d \in \phi_{\text{dst}}(23.1.4.0/24)$ and some arbitrary source address s. The top of the trace given by the semantics $\llbracket \mathcal{N} \rrbracket_i \langle c_1, \langle d, s \rangle \rangle$ is shown in Table 2.

4 Realtime Verification of Layered Networks

Given a network $\mathcal{N} = \langle V, E, \mathcal{H}^*, \mathcal{T}, \mathcal{R} \rangle$ over stacks of headers $\mathcal{H}^*$ and a new rule r being inserted or removed from $\mathcal{R}$, our goal is to incrementally verify the correctness of $\mathcal{N}$ with respect to some user defined properties of interest.

In this section we first give a brief overview of how existing incremental verification algorithms work for finite header sets $\mathcal{H}$ (§4.1). We then show how this notion of equivalence class falls apart for the infinite space of header stacks $\mathcal{H}^*$, which leads existing algorithms to not terminate. To solve this problem, we define a new notion of *partial* equivalence class based on only the top of the header stack (§4.2). Partial equivalence classes can be computed efficiently, however they do not necessarily guarantee equivalent network-wide behavior. Instead, we develop an algorithm that lazily refines these classes while verifying properties (§4.3 and §4.4).

4.1 Existing incremental verifiers

Most incremental data plane verifiers today work by analyzing the network rules $\mathcal{R}$ and, based on that analysis, dividing the headers $\mathcal{H}$ into subsets that have the same forwarding behavior, which can then be checked using graph algorithms. More specifically equivalence classes are defined as:

Definition 4.1 (Trace hops). *Given a trace σ (from §3.3) consisting of located packets L, we define function $\text{hops}(\sigma)$, which produces only the nodes in the trace, inductively over σ as $\text{hops}(\varepsilon) = \varepsilon$ and $\text{hops}(\sigma' \cdot \langle u, h \rangle) = \text{hops}(\sigma') \cdot u$.*

Definition 4.2 (Equivalence classes). *A set of header sets* $\{\mathcal{H}_1,\ldots,\mathcal{H}_n\}$ *are equivalence classes for a network* $\mathcal{N} = \langle V, E, \mathcal{H}, \mathcal{T}, \mathcal{R} \rangle$ *if the following conditions hold:*

- $\mathcal{H} = \mathcal{H}_1 \cup \ldots \cup \mathcal{H}_n$ *(complete)*
- $\forall i, j \in \{1,..,n\}, i \neq j \Rightarrow \mathcal{H}_i \cap \mathcal{H}_j = \emptyset$ *(disjoint)*
- $\forall j \in \{1,..,n\}, \forall h_1, h_2 \in \mathcal{H}_j, \forall u \in V, \forall i \in \mathbb{N},$ *(g-equiv)*
  $\mathrm{hops}(\llbracket \mathcal{N} \rrbracket_i \langle u, h_1 \rangle) = \mathrm{hops}(\llbracket \mathcal{N} \rrbracket_i \langle u, h_2 \rangle)$

Existing incremental verification tools compute an over-approximate set of equivalence classes $\{\mathcal{H}_1,\ldots,\mathcal{H}_n\}$ using intricate data structures such as multi-dimentional tries [29] and Binary Decision Diagrams (BDDs) [46]. While early work on incremental verification such as Veriflow [29] and Deltanet [20] could not handle rule transformations (i.e., all transformations must be τ_{id}), more recent work such as AP [42] and APKeep [46] can account for transformations.

At a high-level, these tools work as follows. First, they compute a set of equivalence classes $\{\mathcal{H}_1,\ldots,\mathcal{H}_n\}$ based on the rule match fields. Next, for each transformation $\tau \in \mathcal{T}$ and each $i \in \{1,..,n\}$, they compute $\tau(\mathcal{H}_i) = \{\tau(h) \mid h \in \mathcal{H}_i\}$. Since the transformed sets may now violate the disjoint condition, the resulting set $\{\mathcal{H}_1,\ldots,\mathcal{H}_n, \tau(\mathcal{H}_1),\ldots,\tau(\mathcal{H}_n)\}$ is made disjoint by dividing up these sets. This process is iterated with all transformations until no more changes occur.

The problem with equivalence classes. There are several problems that occur when trying to lift this approach to equivalence class generation to *stacks* of headers. One problem is that the space of header stacks $\mathcal{H}^*$ is infinite and symbolic data structures in existing tools cannot represent and manipulate infinite sets of values. This is generally a hard problem, which we solve in §4.3.

Even with data structures to manipulate such infinite sets, the algorithm discussed previously does not necessarily terminate in this infinite space. For instance, an equivalence class for stacks with a single header: $\mathcal{H}_i^* = \{\varepsilon \cdot h \mid h \in \mathcal{H}\}$ does not terminate with τ_{push} – one would compute a new equivalence class for packets with two headers, then three, and so on.

4.2 Partial equivalence classes

To solve this problem, we introduce a new notion of *partial* equivalence classes. Partial equivalence classes capture sets of packets that will have the same forwarding behavior at every node in the network but may not be transformed unambiguously by transformations to other partial equivalence classes. Formally, we define them as:

Definition 4.3 (Partial Equivalence Classes). *A set of header sets* $\{\mathcal{H}_1,\ldots,\mathcal{H}_n\}$ *are partial equivalences classes for a network* $\mathcal{N} = \langle V, E, \mathcal{H}, \mathcal{T}, \mathcal{R} \rangle$ *if the following hold:*

- $\mathcal{H} = \mathcal{H}_1 \cup \ldots \cup \mathcal{H}_n$ *(complete)*
- $\forall i, j \in \{1,..,n\}, i \neq j \Rightarrow \mathcal{H}_i \cap \mathcal{H}_j = \emptyset$ *(disjoint)*
- $\forall j \in \{1,..,n\}, \forall h_1, h_2 \in \mathcal{H}_j, \forall u \in V,$ *(l-equiv)*
  $\mathrm{edge}(\Omega(u, h_1)) = \mathrm{edge}(\Omega(u, h_2)) \wedge$
  $\mathrm{modify}(\Omega(u, h_1)) = \mathrm{modify}(\Omega(u, h_2))$

The difference between partial equivalence classes and equivalence classes (Definition 4.2) is subtle. We demonstrate the difference in Figure 3. In the example, packet headers consist of a destination IP field and time-to-live (TTL) field. If we ignore the layering transformations τ_{push} and τ_{pop}, which make the example not terminate, existing tools AP and AP-Keep would compute the equivalence classes shown in Figure 3b according to Definition 4.2. There are 257 equivalence classes. This large number comes from repeatedly applying τ_{ttl} to compute the transitive closure of equivalence classes as described in §4.1. In contrast, there are only 3 partial equivalence classes for the example, shown in Figure 3c since they depend only on local forwarding behavior.

Note that partial equivalence classes do not guarantee equivalent end-to-end behavior of packets, only local forwarding. For instance the packets $\langle 10.7.1.2, 255 \rangle$ and $\langle 10.7.1.2, 1 \rangle$ belong to the same partial equivalence class (2) in Figure 3c. Yet when sent from v_1, the latter packet will be dropped at v_2 while the former will be forwarded to v_3.

Of importance is that the definition of partial equivalence classes depends only on the rule transformations τ applied rather than the application of $\tau(\mathcal{H}_i)$ to some set of packets. This means that we can compute partial equivalence classes efficiently for header stacks using techniques similar to that of prior work [46] by looking only at the top of the stack.
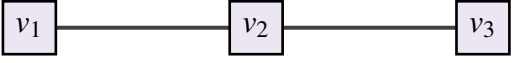
4.3 Verification algorithm overview

Given a set of (changed) partial equivalence classes and a property P, our objective is to check whether P holds for all packets in all of the (changed) partial equivalence classes.

Our approach is as follows: given a set of partial equivalence classes $\{\mathcal{H}_1,\ldots,\mathcal{H}_n\}$ we start by exploring the reachable paths from every source node using a depth-first search. At each node u, packets in the partial equivalence class for $\mathcal{H}_i$ will all have the same next hop v and transformation τ (by definition). We proceed to apply $\tau(\mathcal{H}_i)$ to get some new set of packets $\mathcal{H}_i'$. Because $\mathcal{H}_i'$ may partially overlap with one or more existing partial equivalence classes, we identify all other partial equivalence classes $\mathcal{H}_{i_1},\ldots,\mathcal{H}_{i_m}$ such that $\forall j \in \{1,..,m\}, \mathcal{H}_i' \cap \mathcal{H}_{i_j} \neq \emptyset$. We then continue the search with each subset $(\mathcal{H}_i' \cap \mathcal{H}_{i_j})$.

Representing header stacks. We still have the problem of symbolically representing the infinite space of stacks of headers $\mathcal{H}^*$. To do so, we use a decomposed representation where we model a set of header stacks as a concrete stack of symbolic header sets. For instance, suppose the set of reachable header stacks at a given node is $\{\varepsilon \cdot h_1 \cdot h_2, \ \varepsilon \cdot h_3 \cdot h_4\}$. We instead represent this set as the stack of header sets given by the stack $\varepsilon \cdot \{h_1, h_3\} \cdot \{h_2, h_4\}$.

Of course, this decomposed representation naturally over approximates the set of headers (e.g., it would appear that $\varepsilon \cdot h_1 \cdot h_4$ is a reachable header stack). However, by carefully tracking the transformations that modify the stack (e.g., that

rule	p	e	m	τ
r_0	100	$\langle v_1, v_1 \rangle$	$\phi_{ttl}(\{0\})$	τ_{drop}
r_1	200	$\langle v_1, v_2 \rangle$	$\phi_{dst}(10.7.1.0/24)$	$\tau_{push} \circ \tau_{ttl}$
r_2	300	$\langle v_1, v_2 \rangle$	$\mathcal{H}^*$	τ_{drop}
r_3	100	$\langle v_2, v_2 \rangle$	$\phi_{ttl}(\{0\})$	τ_{drop}
r_4	200	$\langle v_2, v_3 \rangle$	$\phi_{dst}(10.7.1.0/24)$	$\tau_{pop} \circ \tau_{ttl}$
r_5	300	$\langle v_2, v_2 \rangle$	$\mathcal{H}^*$	τ_{drop}
r_6	100	$\langle v_3, v_3 \rangle$	$\phi_{ttl}(\{0\})$	τ_{drop}
r_7	200	$\langle v_3, v_3 \rangle$	$\mathcal{H}^*$	τ_{delv}

(a) Example network topology and rules.

(1)	$\phi_{dst}(10.7.1.0/24)$	$\cap$	$\phi_{ttl}(\{0\})$
(2)	$\phi_{dst}(10.7.1.0/24)$	$\cap$	$\phi_{ttl}(\{1\})$
$\ldots$			$\ldots$
(256)	$\phi_{dst}(10.7.1.0/24)$	$\cap$	$\phi_{ttl}(\{255\})$
(257)	$\mathcal{H} - \phi_{dst}(10.7.1.0/24)$		

(b) Equivalence classes computed by APKeep [46].

(1)		$\phi_{ttl}(\{0\})$	
(2)	$\phi_{dst}(10.7.1.0/24)$	$\cap$	$\phi_{ttl}(\{1,..,255\})$
(3)	$\mathcal{H} - \phi_{dst}(10.7.1.0/24)$	$\cap$	$\phi_{ttl}(\{1,..,255\})$

(c) Partial equivalence classes computed by KATRA.

Figure 3: Running example of computing reachability in a simple multilayer network. Example network has headers consisting of a destination IP and a time-to-live (TTL) field: $h \in \mathcal{H} = \langle d, t \rangle$ where $d \in \{0, .., 2^{32} - 1\}$ and $t \in \{0, .., 255\}$. The transformation τ_{ttl} decrements the TTL field. (a) shows APKeep equivalence classes for the single-layer version of the network and (b) Katra's partial equivalence classes for the multi-layer version.

only h_1 leads to h_2 and only h_3 leads to h_4), this representation remains precise. On the other hand, the decomposed representation is convenient because it allows for modeling arbitrary sized stacks and can execute τ_{push} and τ_{pop} cheaply on the symbolic representation since it is just a concrete stack operation. It also lets us leverage existing efficient data structures such as those based on BDDs, to manipulate the stacks despite not having a fixed size.

Given a decomposed stack of header sets $\sigma = \varepsilon \cdot H_1 \cdot \ldots \cdot H_n$ the usual definitions for τ_{push} and τ_{pop} apply, and we use a definition of a transformation τ applied to stacks: $\tau(\sigma \cdot h) = \sigma \cdot \tau(H)$. One drawback with this definition is that the headers at different layers of the stack lose dependencies between them. For instance, if the stack $\varepsilon \cdot H_1 \cdot H_1$ is filtered and becomes $\varepsilon \cdot H_1 \cdot H_2$, it may be that the new stack should be $\varepsilon \cdot H_2 \cdot H_2$ since only those packets with the inner header in H_2 would have pushed to headers that later survived the filter. In general, we track the transformations applied to the headers at each layer of the stack, and then "repair" the stack on demand whenever our representation is at risk of losing precision.

4.4 Layered verification algorithm

The algorithm for verifying arbitrarily layered networks is shown in Algorithm 1. CheckProperty takes as input the network $\mathcal{N}$, the partial equivalence class (e.g., one that changed after a rule insertion or deletion) $\mathcal{H}_i$, a set of source nodes S, a destination node d, and a path property P to check for each pair of source and destination.

The algorithm starts by running a depth-first search from each source node $s \in S$ (line 7) and tracking the visited nodes. Each node in the algorithm contains (i) a topology node, (ii) the current partial equivalence class (initially $\mathcal{H}_i$), and (iii) the current symbolic stack (initially $\varepsilon \cdot \mathcal{H}_i$). An invariant of the

algorithm is that the top of the symbolic stack is a subset of the current partial equivalence class.

The depth-first search first looks up the next hop (edge and transformation τ on line 11) for the current partial equivalence class $\mathcal{H}_i$ and node u. It then applies the transformation τ to the current stack (line 12). If τ is undefined for this stack, then the trace is terminated and the algorithm checks the property P on the path (stored in the previous pointers starting at u on line 14). If the property fails, it returns a counter example.

Otherwise, the algorithm inspects the new top of the stack σ and finds all new overlapping partial equivalence classes $\mathcal{H}_{i_j}$ (line 16). For each, it computes a new stack σ' (line 17) obtained by restricting the top of the stack to this new partial equivalence class. If the top of the stack changed it then "repairs" the rest of the stack (line 20). We go into this operation in more detail in §4.5. Afterwards, the algorithm creates a new node for the next hop v with the new partial equivalence class $\mathcal{H}_{i_j}$ and the new stack σ' (line 20).

At this point the algorithm marks u as visited (line 22) checks if the new node creates an infinite loop (line 25). The details of this check are complex and are covered in detail in §5.1. Finally, if the new node v has not yet been visited, it recursively calls Dfs from this new node (line 28).

Example. We can see an application of Algorithm 1 in Figure 4. This shows the DFS trace produced for the earlier example network shown in Figure 3 that uses a time-to-live field. The execution is shown for the partial equivalence class $\mathcal{H}_i = (2)$, which corresponds to packets in the set $H_0 = \phi_{dst}(10.7.1.0/24) \cap \phi_{ttl}(\{1, .., 255\})$. Initially the algorithm starts in partial equivalence class (2) with the stack $\varepsilon \cdot H_0$. From here, the algorithm discovers that the next hop is v_2 and the transformation is $\tau_{push} \circ \tau_{ttl}$. The result of applying this transformation to $\varepsilon \cdot H_0$ is two new sets of stacks corre-

Algorithm 1: Reachability for layered networks.

Input: Network $\mathcal{N}$, partial equivalence class $\mathcal{H}_i$,
Source locations S, Property P
Output: Counterexample, or null if none

1 **Procedure** CheckProperty$(\mathcal{N}, \mathcal{H}_i, S, P)$
2   visited $\leftarrow \emptyset$
3   **for** s **in** S **do**
4      $u \leftarrow$ new Node(s, $\mathcal{H}_i$, $\varepsilon \cdot \mathcal{H}_i$)
5   **if** $u \notin$ visited **then**
6   u.previous $\leftarrow$ null
7   trace $\leftarrow$ Dfs$(\mathcal{N}, P, \text{visited}, u, 0)$
8   **if** trace $\neq$ null **return** trace

9   **return** null

10 **Procedure** Dfs$(\mathcal{N}, P, \text{visited}, u, i)$
11    $\langle \text{edge}, \tau \rangle \leftarrow$ Forward$(\mathcal{N}, u.\text{loc}, u.\text{ec})$
12    $\sigma \leftarrow \tau(u.\text{stack})$
13   **if** σ undefined **then**
14   **return** (**if** $P(u)$ **then** null **else** GetTrace(u))
15   nexthops $\leftarrow \emptyset$
16   **for** $\mathcal{H}_{i_j}$ **in** OverlappingEcs$(\text{top}(\sigma))$ **do**
17      $\sigma' \leftarrow \text{bot}(\sigma) \cdot \left(\text{top}(\sigma) \cap \mathcal{H}_{i_j}\right)$
18   **if** $\text{top}(\sigma) \neq \text{top}(\sigma')$ **or** $|\sigma| \neq |\sigma'|$ **then**
19        $\sigma' \leftarrow$ Repair(σ')
20      $v \leftarrow$ new Node(tgt(edge), $\mathcal{H}_{i_j}$, σ')
21   nexthops $\leftarrow$ nexthops $\cup \{\langle \tau, v \rangle\}$
22   visited $\leftarrow$ visited $\cup \{u\}$
23   **for** $\langle \tau, v \rangle$ **in** nexthops **do**
24   v.previous $\leftarrow \langle \tau, u \rangle$
25   **if** HasLoop$(v, \text{visited})$ **then**
26   **return** GetTrace(v)
27   **if** $v \notin$ visited **then**
28   trace $\leftarrow$ Dfs$(\mathcal{N}, P, \text{visited}, v, i+1)$
29   **if** trace $\neq$ null **return** trace

30   **return** null

Algorithm 2: Unbounded loop check.

Input: Graph node u, and visited nodes visited
Output: Boolean for if there is an infinite/finite loop.

1 **Procedure** HasLoop$(u, \text{visited})$
2    $C \leftarrow \{n \mid n \in \text{visited}, n.\text{loc} = u.\text{loc}\}$
3    $c \leftarrow u.\text{previous}$
4    $\mu \leftarrow |u.\text{stack}|$
5   **while** $c \neq$ null **and** $C \neq \emptyset$ **do**
6      $\mu \leftarrow$ Min$(\mu, |c.\text{stack}|)$
7   **if** $c \in C$ **then**
8        $C \leftarrow C - \{c\}$
9        $\gamma \leftarrow$ LCS$(u.\text{stack}, c.\text{stack})$
10   **if** $\mu > |c.\text{stack}| - \gamma$ **then**
11   **return** true
12      $c \leftarrow c.\text{previous}$
13   **return** false

$$H_0 = \phi_{\text{dst}}(10.7.1.0/24) \cap \phi_{\text{ttl}}(\{1,..,255\})$$
$$H_1 = \phi_{\text{dst}}(10.7.1.0/24) \cap \phi_{\text{ttl}}(\{1,..,254\})$$
$$H_2 = \phi_{\text{dst}}(10.7.1.0/24) \cap \phi_{\text{ttl}}(\{0\})$$
$$H_3 = \phi_{\text{dst}}(10.7.1.0/24) \cap \phi_{\text{ttl}}(\{2,..,255\})$$
$$H_4 = \phi_{\text{dst}}(10.7.1.0/24) \cap \phi_{\text{ttl}}(\{1\})$$
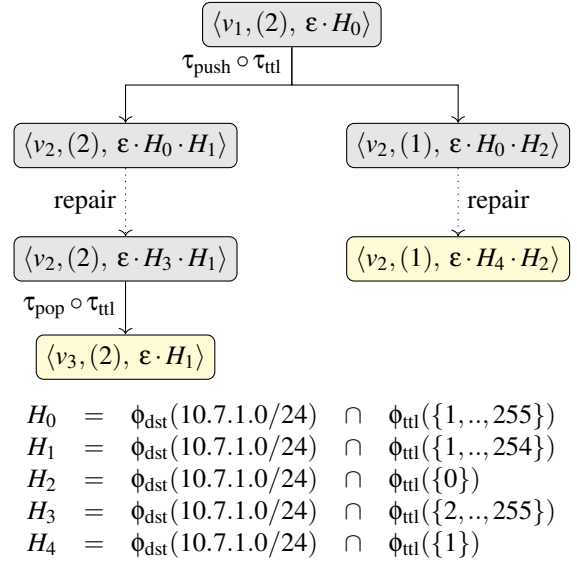
Figure 4: Example execution of Algorithm 1 for the partial equivalence class $\mathcal{H}_i = (2)$ from the example in Figure 3.

sponding to different partial equivalence classes. The first is $\varepsilon \cdot H_0 \cdot H_1$, which remains in partial equivalence class (2). The second is $\varepsilon \cdot H_0 \cdot H_2$, which now falls into partial equivalence class (1) since the TTL field reaches zero. In both cases, we "repair" the stack since the first header may be wrong. The results are given by $\varepsilon \cdot H_3 \cdot H_1$ and $\varepsilon \cdot H_4 \cdot H_2$. Those packets in partial equivalence class (1) are now dropped since the TTL field is 0. And the remaining packets are forwarded to v_3, decapsulated, and eventually delivered.

4.5 Repairing the stack

Recall in the example in Figure 4, the initial state is $\langle v_1, (2), H_0 \rangle$ capturing all packets for the destination pre-

fix with TTL greater than zero. After being transformed by $\tau_{\text{push}} \circ \tau_{\text{ttl}}$, the resulting headers for partial equivalence class (2) are given by the stack $\varepsilon \cdot H_0 \cdot H_1$. Regrettably, H_0 is no longer correct because it contains a packet with a TTL field of 1, which would be 0 after the TTL decrement and thus no longer be part of H_1, which has TTL values in $\{1,..,254\}$.

The problem generally is that after restricting the top of the stack (Algorithm 1, line 17), the bottom of the stack may contain too many headers. To repair the stack, we reverse all transformations applied to the current stack to recover the initial set of packets from the source that will eventually lead to the new restricted stack. We then replay the transformations forward with the correct initial set to simulate the construction

of the repaired stack as though we had started with the set that takes into account the later restriction.

Definition 4.4 (Transformation Inverse). *Given a transformation τ for sets of stacks, we define its inverse as $\tau^{-1}(H^*) = \{\sigma \in \mathcal{H}^* \mid \tau(\sigma) \in H^*\}$.*

Assume we have a sequence of stacks and transformations starting from the initial state of the depth-first search: $\sigma_1 \xrightarrow{\tau_1} \sigma_2 \xrightarrow{\tau_2} \ldots \sigma_{n-1} \xrightarrow{\tau_{n-1}} \sigma_n$. We compute:

$$
\begin{aligned}
\sigma_{\text{init}} &= (\tau_1^{-1} \circ \ldots \circ \tau_n^{-1})(\sigma_n) \\
\sigma_{\text{repair}} &= (\tau_n \circ \ldots \circ \tau_1)(\sigma_{\text{init}})
\end{aligned}
$$

Example. We clarify this idea through an example. In the DFS shown in Figure 4, at node $\langle v_2, (2), H_0 \cdot H_1 \rangle$ we perform a stack repair operation. To do so, we compute σ_{init}:

$$
\begin{aligned}
& \sigma_{\text{init}} && \textit{compute } \sigma_{\text{init}} \\
=\ & (\tau_{\text{push}} \circ \tau_{\text{ttl}})^{-1}(H_0 \cdot H_1) && \textit{unfold definition} \\
=\ & \tau_{\text{push}}^{-1}(\tau_{\text{ttl}}^{-1}(H_0 \cdot H_1)) && \textit{function composition} \\
=\ & \tau_{\text{push}}^{-1}(H_0 \cdot H_3) && \textit{inverse of } \tau_{\text{ttl}} \\
=\ & H_3 && \textit{inverse of } \tau_{\text{push}}
\end{aligned}
$$

$$
\begin{aligned}
& \sigma_{\text{repair}} && \textit{compute } \sigma_{\text{repair}} \\
=\ & (\tau_{\text{push}} \circ \tau_{\text{ttl}})(H_3) && \textit{unfold definition} \\
=\ & \tau_{\text{ttl}}(\tau_{\text{push}}(H_3)) && \textit{function composition} \\
=\ & \tau_{\text{ttl}}(H_3 \cdot H_3) && \textit{definition of } \tau_{\text{push}} \\
=\ & H_3 \cdot H_1 && \textit{definition of } \tau_{\text{ttl}}
\end{aligned}
$$

This result is given by node $\langle v_2, (2), H_3 \cdot H_1 \rangle$ in Figure 4.

4.6 Property expressiveness

For efficiency, our algorithm concerns itself primarily with checking path properties P that are "subpath closed":

Definition 4.5 (Subpath Closed). *A property P is subpath closed if whenever P holds on a sequence of nodes $u_0, \ldots, u_n$, it also holds on any subsequence $u_j, u_{j+1}, \ldots, u_n$ for $j \geq 0$.*

Subpath-closed properties include reachability to a destination, loop-freedom, and network isolation. We focus on this subset of properties because they permit an efficient implementation by avoiding exploring previously visited nodes (Algorithm 1, line 5). However, this is not an inherent limitation of our algorithm – with only minor changes it can be used to check any path properties for packets, albeit at greater cost since we can not reuse previously visited nodes.

5 Algorithm Correctness

We now prove that Algorithm 1 is sound with respect to our concrete packet semantics from §3.3. But first we must define what it means for a DFS state to contain a located packet:

Definition 5.1 (DFS overapproximation). *For a located packet ℓ and Dfs node u, we write $\ell \in u$ if $\ell = \langle v, \sigma \rangle$ and $u.\text{loc} = v$ and $\sigma \in u.\text{stack}$ and $\text{top}(\sigma) \in u.\text{ec}$.*

rule	p	e	m	τ
r_0	100	$\langle u_1, u_1 \rangle$	$\phi_{\text{dst}}(10.0.1.0/24)$	$\tau_{\text{push}} \circ \tau_d \circ \tau_{\text{push}}$
r_1	100	$\langle u_2, u_2 \rangle$	$\phi_{\text{dst}}(d)$	τ_{pop}
r_2	200	$\langle u_2, u_3 \rangle$	$\phi_{\text{dst}}(10.0.1.0/24)$	τ_{push}
r_3	300	$\langle u_2, u_3 \rangle$	$\mathcal{H}^*$	τ_{drop}
r_3	100	$\langle u_3, u_2 \rangle$	$\mathcal{H}^*$	τ_{push}
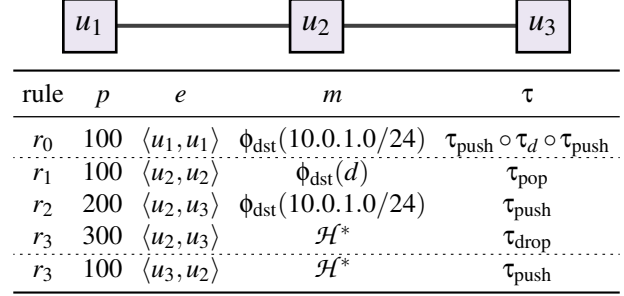
Figure 5: Example network with an infinite loop for the 10.0.1.1 address. The value d can be any other IP address.

Now given this definition, we state the soundness of Algorithm 1 as follows by relating the concrete semantics to the DFS calls made in the algorithm. For the simplicity of the proof, we elide the visited set optimization (lines 5 and 27) and the loop check (line 25). We revisit the loop check for termination in §5.1

Theorem 5.1 (Soundness). *For any network $\mathcal{N}$, partial equivalence class $\mathcal{H}_j$, node v, header $h \in \mathcal{H}_j$, located packet $\ell = \langle v, \varepsilon \cdot h \rangle$, and step $i \geq 0$, if not $\mathcal{N} \otimes \langle i, \ell \rangle$ then after calling $\text{CheckProperty}(\mathcal{N}, \mathcal{H}_j, \{v\}, P)$ there will eventually be a call to $\text{Dfs}(\mathcal{N}, P, \_, u, i)$ for some node u such that $\text{top}(\llbracket \mathcal{N} \rrbracket_i \ell) \in u$.*

Proofs are included as extra material in the appendix.

Corollary 5.1 (Property checking). *If i is the smallest step such that $\mathcal{N} \otimes \langle i, \ell \rangle$ then Algorithm 1 checks $P(u)$ for some DFS node u such that $\text{top}(\llbracket \mathcal{N} \rrbracket_i \ell) \in u$.*

5.1 Infinite Loops and Termination

While Theorem 5.1 says that Algorithm 1 is sound, it says nothing about whether it will terminate. Intuitively, a network $\mathcal{N}$ contains a loop for a header h, whenever that packet will visit a node infinitely often in the future[2]. Catching infinite loops is vital since otherwise Algorithm 1 may not terminate. Finding loops in layered networks is surprisingly challenging since the space of header stacks is infinite and no stack need repeat to have an infinite loop.

We start by defining a loop for a given header:

Definition 5.2 (Network Loop). *Given a network $\mathcal{N}$, an input located packet ℓ induces a loop if there exists a step index $i \in \mathbb{N}$ for the start of the loop such that for all steps $j \in \mathbb{N}$ where $j \geq i$, there exists a future $k \in \mathbb{N}$ such that:*

$$
\begin{aligned}
& (1) && |\llbracket \mathcal{N} \rrbracket_j \ell| < |\llbracket \mathcal{N} \rrbracket_k \ell| \\
& (2) && \text{top}(\text{hops}(\llbracket \mathcal{N} \rrbracket_j \ell)) = \text{top}(\text{hops}(\llbracket \mathcal{N} \rrbracket_k \ell))
\end{aligned}
$$

---

[2]Note: networks modeling the TTL field like in Figure 7 do not have a loop in the algorithmic sense because the packet will eventually expire after a finite number of steps. Such issues can be caught with an appropriate property P that looks for packets that eventually expire with TTL 0.

In other words, a loop exists if beyond some point in the trace (*i*) the trace will continue to grow forever and repeatedly visit the same nodes in the network.

Example. We demonstrate the difficulty of detecting infinite loops in Figure 5. Unlike in single layer networks, loops in multi-layer networks may be transient even when the top of the stack repeats at the same node because of implicit state lower in the header stack. Further, any given stack may not repeat even when an infinite loop exists since the stack can keep growing. Consider a trace for the example in Figure 5 for traffic sent from u_1 with the 10.0.1.1 destination.

$\langle u_1, 10.0.1.1 \rangle$	*encapsulate twice*
$\langle u_2, 10.0.1.1 \cdot d \cdot d \rangle$	*pop*
$\langle u_2, 10.0.1.1 \cdot d \rangle$	*pop*
$\langle u_2, 10.0.1.1 \rangle$	*forward to u_3*
$\langle u_3, 10.0.1.1 \cdot 10.0.1.1 \rangle$	*forward to u_2*
$\langle u_2, 10.0.1.1 \cdot 10.0.1.1 \cdot 10.0.1.1 \rangle$	*forward to u_3*
$\ldots$	

Note that the top of stack d is repeated at node u_2, however, this is not the cause of the infinite loop since eventually this outer header is removed and the forwarding proceeds according to the inner header for the 10.0.1.1 address. Later, however, there is an infinite loop despite the stack never repeating exactly at any node in the trace.

Necessary and sufficient conditions. Suppose we have a current header stack $\sigma \cdot h$ at node u, and later on we arrive at u once more, but with header stack $\sigma \cdot \sigma' \cdot h$ with the same shared prefix σ. Moreover, assume that between visiting u twice, the rules never examine the contents of σ. If these conditions hold then the top of the stack h "regenerates" itself without needing context from σ. In this case, we can infer that there will be an infinite loop at u given by: $\langle u, \sigma \cdot h \rangle \longrightarrow \langle u, \sigma \cdot \sigma' \cdot h \rangle \longrightarrow \langle u, \sigma \cdot \sigma' \cdot \sigma' \cdot h \rangle \longrightarrow \ldots$ This idea is similar to repeating heads from the verification of pushdown systems [36] and we prove that this condition is both sufficient and necessary for a permanent loop:

Theorem 5.2 (Loop conditions). *Given a network $\mathcal{N}$ over $\mathcal{H}^*$, an input ℓ induces a loop if and only if there exists $i, k \in \mathbb{N}$, $\sigma, \sigma' \in \mathcal{H}^*$, and $h \in \mathcal{H}$ such that:*

(1) $\mathrm{top}(\llbracket \mathcal{N} \rrbracket_i \ell) = \langle u, \sigma \cdot h \rangle$
(2) $\mathrm{top}(\llbracket \mathcal{N} \rrbracket_k \ell) = \langle u, \sigma \cdot \sigma' \cdot h \rangle$
(3) $\forall j, \ i < j < k \Rightarrow \exists v, \sigma'', \ \mathrm{top}(\llbracket \mathcal{N} \rrbracket_j \ell) = \langle v, \sigma \cdot \sigma'' \rangle$

Loop detection algorithm. Based on the insights from Theorem 5.2, we develop an efficient procedure for checking loops during traversal, which is described in Algorithm 2. Given the current node (*u*) in the DFS, and the visited nodes (visited) the procedure checks for a loop by looking up all candidate nodes (C) for the same current topology location (*u*.loc, line 2). The algorithm walks backwards through the current path (line 5) and computes the longest common suffix (LCS) γ between the tops of the stacks for the two nodes u

```
1   // instantiate a new network verifier
2   var headerType = new HeaderType(
3       ("dstip", 32), ("srcip", 32));
4   var nv = new NetworkVerifier(headerType);
5
6   // build the network topology
7   var (n1, n2) = nv.GetOrAddNodes("n1", "n2");
8   var (e12, e21) = nv.GetOrAddBiEdge(n1, n2);
9
10  // register the properties we want to monitor
11  nv.AddCheck(new LoopCheck(nv.AllHeaders()));
12
13  // create new prioritized forwarding rules
14  var r = nv.CreateRange(
15      (10, 20), (0, uint.MaxValue));
16  var t = nv.Seq(nv.Push(), nv.Set("dstip", 10));
17  var rule1 = new Rule(100, e12, r, t);
18  var rule2 = new Rule(100, e21, r, nv.Pop());
19
20  // find violations from adding rules.
21  var violations1 = nv.AddRule(rule1);
22  var violations2 = nv.AddRule(rule2);
23  Assert.AreEqual(1, violations2.Count);
```

Figure 6: Example use of the KATRA verification API.

and c (line 9) while also tracking the minimum stack size μ between the two nodes. If μ is greater than $|c.\mathrm{stack}| - \gamma$ (where γ generalizes h in the loop conditions) then there is a loop (line 11). If the set of candidate loop nodes has been exhausted, the algorithm terminates early.

6 Implementation

We have built an incremental verification system, KATRA for layered networks based on the idea presented. KATRA is implemented as a C# library and is written in around 8K lines of code. KATRA's implementation for computing header equivalence classes is based on the algorithm from [46], but is modified to incrementally compute the *minimal* set of partial equivalence classes (see §4.2). An example of an API for the tool is shown in Figure 6. The tool is programmable and is parameterized by the format of the header (e.g., MPLS vs. IPv4) that the user wants to check (line 1)[3]. Our implementation extends §3.2 to support mulipath routing.

Optimizations. KATRA makes use of several optimizations to scale. One key challenge is that the use of partial equivalence classes (§4.2) means that we must find overlapping equivalence classes during traversal (Algorithm 1, line 16). To make this operation fast, for every packet set H we keep a pair of $\langle b, H \rangle$ where H is the set itself modeled as a BDD [10], and b is a multi-dimensional bounding box that overapproximates

---

[3]To model different headers in different layers (e.g., Ethernet and IPv4 headers), one can define a "master" header with the union of fields across headers along with a field indicating which header is currently being used.
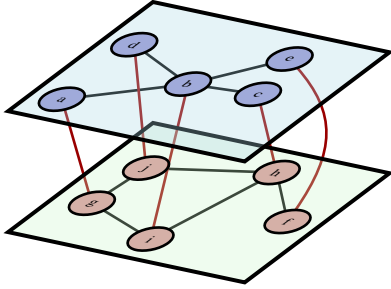
Figure 7: Example of a layered benchmark network with layers $\ell = 2$ and nodes $n = 5$.

the set of headers in H. When sets are unioned or intersected, the corresponding bounding boxes are grown or shrunk to remain safe overapproximations for the sets of headers.

Keeping bounding boxes for header sets allows for the use of fast collision detection data structures. We leverage bounding volume hierarchies [40], which are hierarchical balanced trees of bounding volumes used in game engines to quickly eliminate possible collisions.

Our implementation also examines header sets H and determines if the fields lie on prefix boundaries (e.g., for IPv4 prefix-based forwarding). If so, it uses an optimized trie data structure to accelerate the collision detection.

7 Evaluation

We are primarily interested in evaluating the performance of KATRA relative to a straightforward extension of prior work that models packets with a fixed (bounded) number of headers N. Of course, this approach requires a user to specify N and may be unsound when N is not large enough to handle the maximum stack possible in the network. However, if N is chosen carefully this provides a reasonable comparison point.

7.1 Different implementations

To compare the approach in KATRA with that of duplicate headers (DUP), we instantiate our framework (§6) with two types of headers. For DUP, we instantiate the verifier with a header that is similar to that of Figure 6 (line 2) but extended to a full IPv4 header, and replicated N times. We choose N to account for the maximum amount of layering in each benchmark and do not evaluate DUP on networks that contain unbounded loops, since it will give incorrect results.

Each field in the DUP header has versions f_1 to f_N and f_1 represents the outermost header (top of stack). The push operation is implemented by copying each field f_i to f_{i+1}, its next layer version, and the bottom header is lost in the process if the stack exceeds size N. The pop operation is implemented similarly by copying each field f_{i+1} to f_i.

Single layer performance. APKeep was demonstrated to outperform prior incremental verifiers while also being more

robust to multi-dimensional rules [46]. However, since APKeep is not open source, we instead use our implementation of KATRA, which uses a similar base algorithm to compare results. We ran KATRA on the same datasets reported on in the APKeep paper and originally released by Deltanet [20]. We found the performance for these single layer networks to be similar to the times reported on by APKeep, and as such do not report on the results here. Since the implementation performance is comparable, going forward we report only the times from different instantiations of KATRA.

Moreover, instantiating DUP in KATRA allows us to directly compare our algorithm to a naive solution without other factors coming into play. For example, DUP also makes use of our partial equivalence class reduction, our fast collision detection data structure, and other optimizations.

7.2 Performance on multilayer networks

To measure the performance of KATRA for multilayer networks (i.e., with stack size greater than 1), we generated a parameterized set of benchmark networks.

Benchmark description. The benchmarks have two parameters: the number of layers ℓ in the network, and the number of nodes per layer n. The first layer represents the physical network, while each layer $i > 1$ represents an overlay network built on top of layer $i-1$. Each link in the layer i in the network is implemented by encapsulating a packet and forwarding it according to the destination prefix for the tunnel endpoint in layer $i-1$. Routing in each layer is configured to announce and propagate routes along shortest paths. For each $\langle \ell, n \rangle$ pair, we generate the topologies as random connected graphs and map nodes in each overlay to nodes in the underlay for the purpose of establishing tunnel endpoints.

An example of such a network with $\ell = 2$ and $n = 5$ is shown in Figure 7. In the example, to forward traffic between layer 2 nodes b and e, traffic is encapsulated and forwarded from i to f via h in layer 1. For such networks, there are a total of $O(\ell \cdot n^2)$ forwarding rules.

The first property we check is reachability between all source and destination nodes in the outermost layer ℓ for all advertised subnets. This strategy forces KATRA to reason about the forwarding behavior at every single layer. Because these reachability properties are violated while tunnels are being established at different layers, for this benchmark we disable property checking while connectivity is not expected.

Performance of KATRA ***compared to*** DUP. We show the total verification time of KATRA and DUP in Figure 8 and Figure 9. Figure 8 shows the total time spent recomputing partial equivalence classes for both approaches. KATRA is faster than DUP because DUP must represent significantly larger headers in order to capture the full stack. This leads to larger packet set representations in the BDD library and more expensive set and transform operations.

Similarly, Figure 9 shows the total time spent checking
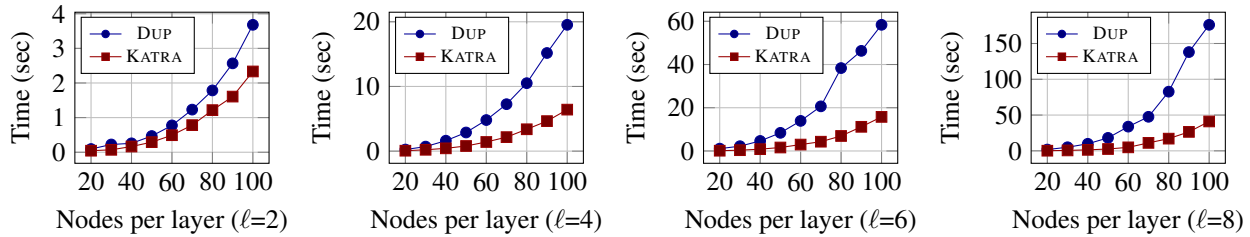
Figure 8: Total time spent recomputing partial equivalence classes for an approach based on duplicate headers DUP vs. KATRA.
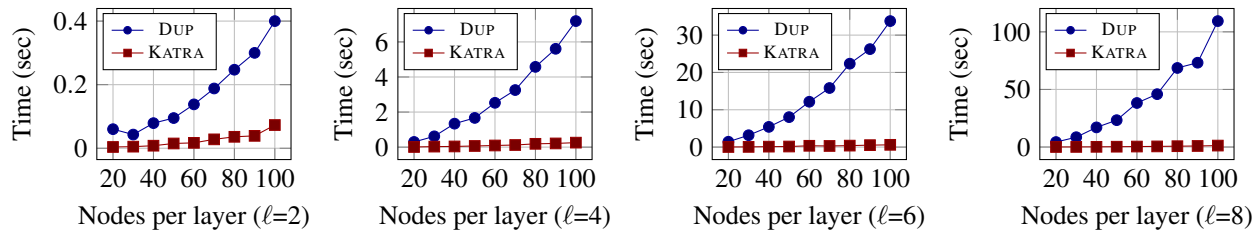
Figure 9: Total time spent checking end-to-end reachability for an approach based on duplicate headers DUP vs. KATRA.

reachability for KATRA and DUP. There is a similar trend, with KATRA's decomposed stack set representation leading to a large speedup over that of DUP. In both cases, the speedup of KATRA grows with both the number of layers ℓ and the number of nodes per layer n in the graph. For instance, at $\ell = 6$ and $n = 100$, property checking is nearly 50x faster.

KATRA *rule update time*. The total verification time grows quickly in part because the number of rules needed to implement the network design is proportional to the square of the size of the network. However, looking at the time for each individual rule update in Figure 10, essentially all updates execute in under 1ms. The graphs show the CDF for rule insertion time in milliseconds. In particular, the insertion time is relatively independent of ℓ yet increases slightly with n.

7.3 Performance of loop checking

To evaluate the performance of Algorithm 2, we used example networks with $\ell = 2$ and replaced the reachability checks for each destination subnet with a single loop check for all packets. Unlike with reachability, this property gets rechecked after every single rule insertion. Since each link in layer 2 crosses many of the same previous nodes in layer 1, this forces Algorithm 2 to check for potential loops frequently.

A CDF of the rule insertion and loop checking time for each update are shown in Figure 11. We vary n from 20 to 80 in increments of 20 and compare the results. Figure 11a shows the checking time when rules are inserted in an arbitrary order. The time grows with the size of the network and can become high at the tail (e.g., around 40ms).

The reason why is that if a rule r with transformation τ_{pop} is inserted early, then every other rule insertion will affect the partial equivalence class for r. In other words, after a decapsulation a packet may now be in any partial equivalence
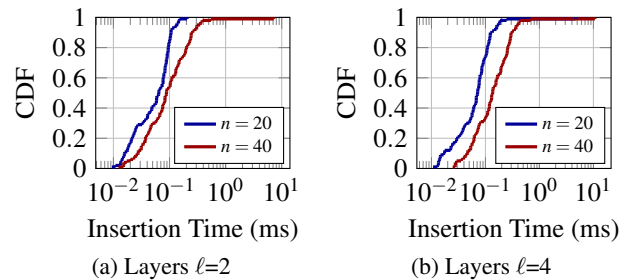
Figure 10: CDF of rule insertion time for (a) ℓ=2 and (b) ℓ=4. Both show results for nodes per layer $n = 20$ and $n = 40$.

class, so when any other partial equivalence class changes, the partial equivalence class for r must also be rechecked. At the extreme, this means that every rule insertion can require rechecking the entire network from scratch. This is inherent in the problem and is not unique to KATRA (e.g., APKeep suffers a similar blowup for these networks).

However, by slightly reordering rule updates, we can improve the performance significantly. In Figure 11b, we show the same results but where the rule insertion order is done in a way to delay the insertion of decapsulation rules. From the figure, we can see that in the latter case, the checking time remains well below 1ms for nearly all rules.

Since this benchmark requires checking the loop property after *all* rule updates, the performance improvement of KATRA grows substantially over that of DUP. Figure 12 shows the total time to verify the loop-free property for all rules updates. It shows the performance for $\ell = 2$ layers where we cap the total verification time at 4 minutes. DUP times out after $n = 100$ with 20K rules while KATRA can verify networks up to $n = 300$ with 180K rules. The relative speedup of KATRA over DUP for $n = 20$ to $n = 100$ is shown in Figure 12b.

(a) Random order (b) Delayed decap
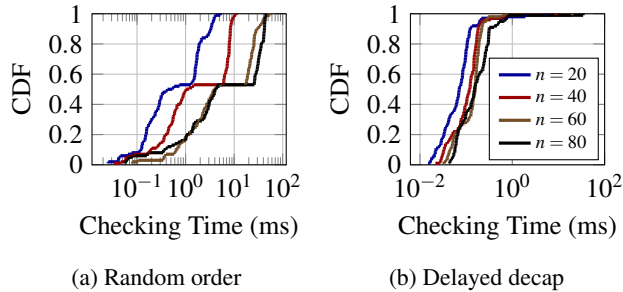
Figure 11: CDF of rule insertion time for $\ell=2$ and $n = 20$ to $n = 80$ in increments of 20. Total checking time for loops is low when decapsulation rule insertion is delayed.

(a) Total time (b) Relative speedup
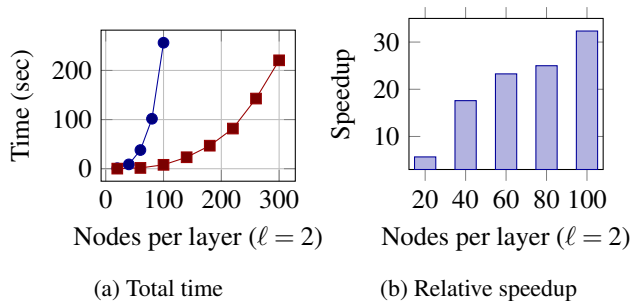
Figure 12: (a) Total verification time for DUP (blue) vs. KATRA (red) for $\ell = 2$ when checking for forwarding loops on every change. (b) Speedup ranges from 5x to 32x.

8 Related Work

KATRA is related to several threads of prior work:

Data plane verification. There is a long line of work on data plane verification, starting from the seminal work of Xie et. al. [27, 41, 42], and incremental verification starting with Veriflow [20, 26, 29, 46]. Most work on data plane verification has assumed stateless and transformation-free forwarding, with the exception of AP [42] and APKeep [46], which handle transformations (see §4). However, none of these works consider *layered* networks where encapsulation and decapsulation are pervasive. AP and APKeep can model finite header stacks (e.g., DUP from §7) but this approach can be unsound and can have poor performance, particularly when encapsulation is common. KATRA builds on prior work to enable incremental verification with transformations and layering.

Layered network verification. There has been little work on verifying multilayer networks. One related work in this area is Tiramisu [2], which can verify some combinations of layer 2 and 3 control plane routing protocols (e.g., BGP, iBGP, OSPF). However, Tiramisu is only superficially related to KATRA: (i) Tiramisu verifies control plane routing while KATRA verifies data plane forwarding, (ii) Tiramisu focuses on specific layering mechanisms (e.g., between iBGP and eBGP) while KATRA focuses on arbitrarily layered *data planes*, and (iii) KATRA is interested in *real-time* (millisecond) verifica-

tion time for incremental changes.

Recent works on verifying MPLS label switching with fast failover [22–24] were the first to leverage the insight that label-based forwarding can be viewed as pushdown automata. The works use polynomial time algorithms to answer reachability questions for all possible failures using overapproximation. While they focus on reasoning about failures, we similarly leverage this insight that ideas from pushdown automata are useful for reasoning about stacks of headers. We generalize this reasoning from concrete label-based forwarding to symbolic forwarding (e.g., prefix-based forwarding) and also focus on *realtime* verification for changes.

There are significant differences in the actual algorithms. These prior works use saturation-based procedures to iteratively compute automata representations of (backward or forward) reachable configurations of the pushdown system. In contrast, our algorithm is an on-the-fly depth-first search over *symbolic* configurations, which include (partial) equivalence classes over the header space.

One work [24] considers abstractions based on network labels to reduce PDS size and proposes a CEGAR-style refinement procedure, which improves performance in many practical examples. Our symbolic configurations are also *abstractions* of the network state space, where the control state is a partial equivalence class in the header space located at a particular node in the network, and the stack is a word over these classes. These abstractions are refined lazily on-the-fly in our novel method for stack repair, such that any trace in our algorithm follows the specified network semantics.

Model checking of pushdown systems. More broadly, our work builds on prior work in model checking of pushdown systems [6, 9, 36], which can naturally represent sequential programs with recursive procedures. Similar to symbolic procedures for pushdown systems [12, 36], we also utilize BDDs [11] for efficient representation of the state space and use a notion similar to *repeating heads* [36] for detecting loops. However, rather than computing sets of reachable configurations, our procedure performs on-the-fly verification to soundly check reachability of located packets.

9 Conclusion

In this paper we have presented KATRA, the first real-time verifier for *layered* networks. KATRA extends incremental data-plane verification to the setting with unbounded header stacks. To do so, we introduced a new network model for layered networks and presented an efficient algorithm for such networks. The algorithm leverages a new idea of *partial* equivalence classes and keeps a decomposed symbolic stack representation that it lazily "repairs" as needed. Comparing KATRA against a solution based on header duplication, we showed that KATRA is 5x-32x faster for just 2 layers, and that its benefits grow with network size and layering.

References

[1] 3GPP. General Packet Radio Service (GPRS); GPRS Tunnelling Protocol (GTP) across the Gn and Gp interface, January 1999.

[2] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. Tiramisu: Fast multilayer network verification. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 201–219, Santa Clara, CA, February 2020. USENIX Association.

[3] Amazon. Amazon ec2 secure and resizable compute capacity to support virtually any workload. https://aws.amazon.com/ec2/, 2021.

[4] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. Netkat: Semantic foundations for networks. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 113–126, New York, NY, USA, 2014. ACM.

[5] John Backes, Sam Bayless, Byron Cook, Catherine Dodge, Andrew Gacek, Alan J. Hu, Temesghen Kahsai, Bill Kocik, Evgenii Kotelnikov, Jure Kukovec, Sean McLaughlin, Jason Reed, Neha Rungta, John Sizemore, Mark Stalzer, Preethi Srinivasan, Pavle Subotić, Carsten Varming, and Blake Whaley. Reachability analysis for aws-based networks. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 231–241, Cham, 2019. Springer International Publishing.

[6] Thomas Ball and Sriram K. Rajamani. Bebop: a path-sensitive interprocedural dataflow engine. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering (PASTE)*, pages 97–103. ACM, 2001.

[7] Nikolaj Bjørner, Garvit Juniwal, Ratul Mahajan, Sanjit A. Seshia, and George Varghese. ddnf: An efficient data structure for header spaces. In *Haifa Verification Conference*, 2016.

[8] J. Border, M. Kojo, J. Griner, G. Montenegro, and Z. Shelby. Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations. Internet Request for Comments, June 2001.

[9] Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR '97: Concurrency Theory, Proceedings*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 1997.

[10] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a bdd package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, DAC '90, pages 40–45, New York, NY, USA, 1990. ACM.

[11] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[12] Javier Esparza and Stefan Schwoon. A BDD-based model checker for recursive programs. In *Computer Aided Verification, International Conference, CAV, Proceedings*, volume 2102 of *Lecture Notes in Computer Science*, pages 324–336. Springer, 2001.

[13] FlexiWAN. The world's first open source sd-wan & sase. https://flexiwan.com/, 2021.

[14] Klaus-Tycho Foerster, Yvonne-Anne Pignolet, Stefan Schmid, and Gilles Tredan. Local fast failover routing with low stretch. *SIGCOMM Comput. Commun. Rev.*, 48(1):35–41, apr 2018.

[15] B. Gleeson, A. Lin, J. Heinanen, Telia Finland, G. Armitage, and A. Malis. A Framework for IP Based Virtual Private Networks. Internet Request for Comments, February 2000.

[16] Google. Google cloud: Cloud computing services. https://cloud.google.com/, 2021.

[17] Google. Network intelligence center: Connectivity tests overview. https://cloud.google.com/network-intelligence-center/docs/connectivity-tests/concepts/overview, 2021.

[18] S. Hanks, Ltd. NetSmiths, T. Li, D. Farinacci, and P. Traina. Generic Routing Encapsulation (GRE). Internet Request for Comments, October 1994.

[19] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan. *SIGCOMM Comput. Commun. Rev.*, 43(4):15–26, August 2013.

[20] Alex Horn, Ali Kheradmand, and Mukul Prasad. Delta-net: Real-time network verification using atoms. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 735–749, Boston, MA, March 2017. USENIX Association.

[21] Karthick Jayaraman, Nikolaj Bjorner, Jitu Padhye, Amar Agrawal, Ashish Bhargava, Paul-Andre C Bissonnette, Shane Foster, Andrew Helwer, Mark Kasten, Ivan Lee, Anup Namdhari, Haseeb Niaz, Aniruddha Parkhi, Hanukumar Pinnamraju, Adrian Power, Neha Milind Raje, and Parag Sharma. Validating datacenters at scale.

In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, pages 200–213, New York, NY, USA, 2019. ACM.

[22] Jesper Stenbjerg Jensen, Troels Beck Krøgh, Jonas Sand Madsen, Stefan Schmid, Jiří Srba, and Marc Tom Thorgersen. P-rex: Fast verification of mpls networks with multiple link failures. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18, page 217–227, New York, NY, USA, 2018. Association for Computing Machinery.

[23] Peter Gjøl Jensen, Dan Kristiansen, Stefan Schmid, Morten Konggaard Schou, Bernhard Clemens Schrenk, and Jiří Srba. Aalwines: A fast and quantitative what-if analysis tool for mpls networks. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '20, page 474–481, New York, NY, USA, 2020. Association for Computing Machinery.

[24] Peter Gjøl Jensen, Stefan Schmid, Morten Konggaard Schou, Jirí Srba, Juan Vanerio, and Ingo van Duijn. Faster pushdown reachability analysis with applications in network verification. In *Automated Technology for Verification and Analysis (ATVA), Proceedings*, volume 12971 of *Lecture Notes in Computer Science*, pages 170–186, 2021.

[25] Andrzej Kamisiński. Evolution of ip fast-reroute strategies. In *2018 10th International Workshop on Resilient Networks Design and Modeling (RNDM)*, pages 1–6, 2018.

[26] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real time network policy checking using header space analysis. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 99–111, Lombard, IL, April 2013. USENIX Association.

[27] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 113–126, San Jose, CA, April 2012. USENIX Association.

[28] S. Kent and K. Seo. Security Architecture for the Internet Protocol. Internet Request for Comments, August 2005.

[29] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 15–27, Lombard, IL, 2013. USENIX.

[30] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright. Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks. Internet Request for Comments, August 2014.

[31] Microsoft. Microsoft azure: Cloud computing services. https://azure.microsoft.com/en-us/, 2021.

[32] J. Moy. Open Shortest Path First Protocol Version 2. Internet Request for Comments, April 1998.

[33] A. Atlas P. Pan, G. Swallow. Fast Reroute Extensions to RSVP-TE for LSP Tunnels. Internet Request for Comments, May 2005.

[34] C. Perkins. IP Encapsulation within IP. Internet Request for Comments, July 1996.

[35] E. Rosen, A. Viswanathan, and R. Callon. Multiprotocol Label Switching Architecture. Internet Request for Comments, January 2011.

[36] Stefan Schwoon. *Model checking pushdown systems*. PhD thesis, Technical University Munich, 2002.

[37] Oliver Spatscheck. Layers of success. *IEEE Internet Computing*, 17(1):3–6, 2013.

[38] 3GPP The Mobile Broadband Standard. 3gpp a global initiative. https://www.3gpp.org/, 2021.

[39] Bingchuan Tian, Xinyi Zhang, Ennan Zhai, Hongqiang Harry Liu, Qiaobo Ye, Chunsheng Wang, Xin Wu, Zhiming Ji, Yihong Sang, Ming Zhang, Da Yu, Chen Tian, Haitao Zheng, and Ben Y. Zhao. Safely and automatically updating in-network acl configurations with intent language. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 214–226. Association for Computing Machinery, 2019.

[40] Ingo Wald. On fast construction of sah-based bounding volume hierarchies. In *2007 IEEE Symposium on Interactive Ray Tracing*, pages 33–40, 2007.

[41] G. G. Xie, Jibin Zhan, D. A. Maltz, Hui Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford. On static reachability analysis of ip networks. In *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, volume 3, pages 2170–2183 vol. 3, March 2005.

[42] Hongkun Yang and Simon S. Lam. Real-time verification of network properties using atomic predicates. *IEEE/ACM Trans. Netw.*, 24(2):887–900, April 2016.

[43] Kok-Kiong Yap, Murtaza Motiwala, Jeremy Rahe, Steve Padgett, Matthew Holliman, Gary Baldus, Marcus Hines, Taeeun Kim, Ashok Narayanan, Ankur Jain, Victor Lin, Colin Rice, Brian Rogan, Arjun Singh, Bert Tanaka, Manish Verma, Puneet Sood, Mukarram Tariq, Matt Tierney, Dzevad Trumic, Vytautas Valancius, Calvin Ying, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Taking the edge off with espresso: Scale, reliability and programmability for global internet peering. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 432–445, New York, NY, USA, 2017. Association for Computing Machinery.

[44] Pamela Zave and Jennifer Rexford. The compositional architecture of the internet. *Commun. ACM*, 62(3):78–87, February 2019.

[45] Hongyi Zeng, Shidong Zhang, Fei Ye, Vimalkumar Jeyakumar, Mickey Ju, Junda Liu, Nick McKeown, and Amin Vahdat. Libra: Divide and conquer to verify forwarding tables in huge networks. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 87–99, Seattle, WA, April 2014. USENIX Association.

[46] Peng Zhang, Xu Liu, Hongkun Yang, Ning Kang, Zhengchang Gu, and Hao Li. Apkeep: Realtime verification for real networks. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 241–255, Santa Clara, CA, February 2020. USENIX Association.

Appendix

Theorem 5.1 (Soundness). *For any network $\mathcal{N}$, partial equivalence class $\mathcal{H}_j$, node v, header $h \in \mathcal{H}_j$, located packet $\ell = \langle v, \varepsilon \cdot h \rangle$, and step $i \geq 0$, if not $\mathcal{N} \otimes \langle i, \ell \rangle$ then after calling* CheckProperty$(\mathcal{N}, \mathcal{H}_j, \{v\}, P)$ *there will eventually be a call to* Dfs$(\mathcal{N}, P, \_, u, i)$ *for some node u such that* top$(\llbracket \mathcal{N} \rrbracket_i \ell) \in u$.

Proof. The proof is by induction on the step i. For the sake of simplicity, we assume that lines 5 and 27 of Algorithm 1, which are optimizations using the visited set, are removed for the remainder of the proof.

Base case ($i = 0$) By assumption we have $\ell = \langle v, \varepsilon \cdot h \rangle$. From unfolding the definition of the semantics $\llbracket \mathcal{N} \rrbracket$ for the ($i = 0$) step, we obtain the following equality:
$$\text{top}(\llbracket \mathcal{N} \rrbracket_0 \ell) = \text{top}(\varepsilon \cdot \ell) = \ell = \langle v, \varepsilon \cdot h \rangle$$
Thus we must prove that there is a call to Dfs$(\mathcal{N}, P, \_, u, 0)$ such that $u.\text{loc} = v$ and $\varepsilon \cdot h \in u.\text{stack}$ and top$(\varepsilon \cdot h) \in u.\text{ec}$. This trivially follows from line 7 of Algorithm 1. Since $S = \{v\}$ (line 3), we see that $s = v$ (line 3) and therefore $u.\text{loc} = v$ as expected, and $u.\text{stack} = \varepsilon \cdot \mathcal{H}_j$ (line 4), which implies that $\varepsilon \cdot h \in u.\text{stack}$ since $\varepsilon \cdot h \in \varepsilon \cdot \mathcal{H}_j \iff h \in \mathcal{H}_j$ by definition and this is an assumption. Finally, we have that top$(\varepsilon \cdot h) = h \in u.\text{ec}$ or $h \in \mathcal{H}_j$ again by assumption.

Inductive case ($i > 0$) The proof proceeds by using the inductive hypothesis for step $i - 1$ to prove that the statement holds for step i. We list out our assumptions from the proof statement as well as the induction hypothesis below:

- not $\mathcal{N} \otimes \langle i, \ell \rangle$
- not $\mathcal{N} \otimes \langle i - 1, \ell \rangle$
- top$(\llbracket \mathcal{N} \rrbracket_i \ell) = \langle v_1, \sigma_1 \rangle$
- top$(\llbracket \mathcal{N} \rrbracket_{i-1} \ell) = \langle v_2, \sigma_2 \rangle$
- there was a call to Dfs$(\mathcal{N}, P, \_, u_2, i - 1)$ for some u_2
- $u_2.\text{loc} = v_2$
- $\sigma_2 \in u_2.\text{stack}$
- top$(\sigma_2) \in u_2.\text{ec}$

Given these assumptions, we must prove that each of the following statements holds as a result:

- there is a call to Dfs$(\mathcal{N}, P, \_, u_1, i)$ for some u_1
- $u_1.\text{loc} = v_1$
- $\sigma_1 \in u_1.\text{stack}$
- top$(\sigma_1) \in u_1.\text{ec}$

We walk through the lines of code in Algorithm 1 starting from the call to Dfs$(\mathcal{N}, P, \_, u_2, i - 1)$ that we know must have taken place. By our assumption that top$(\sigma_2) \in u_2.\text{ec}$, and from the definition of a partial equivalence class (same local forwarding for all packets in the equivalence class), we know the $\langle \text{edge}, \tau \rangle$ pair returned in line 11 must be equivalent to those of the semantics: $\tau = \text{modify}(\Omega(v_2, \sigma_2))$ and $\text{edge} = \text{edge}(\Omega(v_2, \sigma_2))$ from the semantic definition in §3.3. Evaluating $\llbracket \mathcal{N} \rrbracket_i \ell$ there are two cases:

Case 1: if $\tau(\sigma_2)$ is undefined, then we compute: $\llbracket \mathcal{N} \rrbracket_i \ell = \llbracket \mathcal{N} \rrbracket_{i-1} \ell$ and we observe that $\mathcal{N} \otimes \langle i - 1, \ell \rangle$. In this case, the algorithm executes line 14 and terminates. Note that we do not call Dfs again, however, in this case the semantics were terminated at step $i - 1$ which contradicts the assumptions. Further, note that this is the minimal time step i at which $\mathcal{N} \otimes \langle i, \ell \rangle$ since we assumed not $\mathcal{N} \otimes \langle i - 1, \ell \rangle$.

Case 2: if $\tau(\sigma_2)$ is defined, then we compute
$$\langle v_1, \sigma_1 \rangle = \text{top}(\llbracket \mathcal{N} \rrbracket_i \ell) = \text{top}(\llbracket \mathcal{N} \rrbracket_{i-1} \ell \cdot \langle \text{tgt}(\text{edge}), \tau(\sigma_2) \rangle) = \langle \text{tgt}(\text{edge}), \tau(\sigma_2) \rangle$$

By the definition of τ lifted to sets, we know that because $\sigma_2 \in u.\text{stack}$ then it follows that $\tau(\sigma_2) \in \tau(u.\text{stack})$ (line 12) and therefore $\sigma_1 \in \tau(u.\text{stack})$. The algorithm proceeds on line 16 to iterate over all partial equivalence classes that can intersect $\tau(u.\text{stack})$. Because partial equivanence classes are disjoint and complete (see §4.2), there will be exactly one such $\mathcal{H}_{j_k}$ such that top$(\sigma_1) \in \mathcal{H}_{j_k}$. From this we can deduce line 17 will compute a new set of stacks σ' that must contain σ_1 – that is $\sigma_1 \in \sigma'$ by construction.

Line 19 of the algorithm updates σ' as Repair(σ'). Because $\sigma_1 \in \sigma'$ we must show that that $\sigma_1 \in \text{Repair}(\sigma')$ as well. To compute Repair(σ') we first compute $(\tau_1^{-1} \circ \ldots \circ \tau_n^{-1})(\sigma')$, which is equivalent to $\sigma_{\text{init}} = \{\sigma'' \mid (\tau_n \circ \ldots \circ \tau_1)(\sigma'') \in \sigma'\}$. Since σ_1 is the result

of applying $(\tau_n \circ \ldots \circ \tau_1)$ to the initial header $\varepsilon \cdot h$, it follows that $\varepsilon \cdot h \in \sigma_{\text{init}}$. Because $\text{Repair}(\sigma') = \sigma_{\text{repair}} = (\tau_n \circ \ldots \circ \tau_1)(\sigma_{\text{init}})$ and because $\varepsilon \cdot h \in \sigma_{\text{init}}$, it follows that $(\tau_n \circ \ldots \circ \tau_1)(\varepsilon \cdot h) \in (\tau_n \circ \ldots \circ \tau_1)(\sigma_{\text{init}})$ and therefore $(\tau_n \circ \ldots \circ \tau_1)(\varepsilon \cdot h) \in \text{Repair}(\sigma')$.

Finally from lines 20 and 21 a new nexthop is added to the set of nexthops that contains the node u_1 where $u_1.\text{loc} = \text{tgt}(\text{edge})$ and $u_1.\text{ec} = \mathcal{H}_{j_k}$ and $u_1.\text{stack} = \sigma'$. Line 23 iterates over the nexthops and calls Dfs on line 28 with this new node.

To complete the proof, we put together the pieces to show that the 4 conditions above hold.

- line 28 calls $\text{Dfs}(\mathcal{N}, P, \_, u_1, i)$ for the u_1 described previously
- we know that $u_1.\text{loc} = \text{tgt}(\text{edge}) = v_1$
- we know that $\sigma_1 \in u_1.\text{stack}$ because $\sigma_1 \in \sigma'$ and $\sigma' = u_1.\text{stack}$
- we know that $\text{top}(\sigma_1) \in u_1.\text{ec}$ because $\text{top}(\sigma_1) \in \mathcal{H}_{j_k}$ and $u_1.\text{ec} = \mathcal{H}_{j_k}$

$\square$

Corollary 5.1 (Property checking). *If i is the smallest step such that $\mathcal{N} \otimes \langle i, \ell \rangle$ then Algorithm 1 checks $P(u)$ for some DFS node u such that $\text{top}(\llbracket \mathcal{N} \rrbracket_i \ell) \in u$.*

Proof. The proof follows directly from Theorem 5.1. At the $i - 1$ step, we know that there must have been a call to $\text{Dfs}(\mathcal{N}, P, \_, u, i - 1)$ for some u such that $\text{top}(\llbracket \mathcal{N} \rrbracket_{i-1} \ell) \in u$ From the proof we can see that the algorithm will proceed to line 14, where it will check $P(u)$. $\square$

Theorem 5.2 (Loop conditions). *Given a network $\mathcal{N}$ over $\mathcal{H}^*$, an input ℓ induces a loop if and only if there exists $i, k \in \mathbb{N}$, $\sigma, \sigma' \in \mathcal{H}^*$, and $h \in \mathcal{H}$ such that:*

$$
\begin{aligned}
&(1) \quad \text{top}(\llbracket \mathcal{N} \rrbracket_i \ell) = \langle u, \sigma \cdot h \rangle \\
&(2) \quad \text{top}(\llbracket \mathcal{N} \rrbracket_k \ell) = \langle u, \sigma \cdot \sigma' \cdot h \rangle \\
&(3) \quad \forall j, \, i < j < k \Rightarrow \exists v, \sigma'', \, \text{top}(\llbracket \mathcal{N} \rrbracket_j \ell) = \langle v, \sigma \cdot \sigma'' \rangle
\end{aligned}
$$

Proof. First, we require that no rule transformations τ ever both pop and push in the same transformation. For instance, the transformation $\tau_{\text{pop}} \circ \tau_{\text{push}}$ is disallowed, whereas $\tau_{\text{push}} \circ \tau_{\text{push}}$ is allowed. Note that this does not change the expressive power of KATRA since one can always separate such a transformation into multiple transformations across nodes to get the same effect.

Sufficient ($\Leftarrow$) Assume that the conditions (1), (2), and (3) above hold. We must prove that ℓ induces a loop. From (1) and (2), we know that there is a trace for $\llbracket N \rrbracket_k \ell$ to step k of the form:

$$
\underbrace{\langle u_1, \sigma_1 \rangle}_{\text{step 1}} \to \underbrace{\langle u_2, \sigma_2 \rangle}_{\text{step 2}} \to \ldots \to \langle u_{i-1}, \sigma_{i-1} \rangle \to \underbrace{\langle u, \sigma \cdot h \rangle}_{\text{step } i} \to \underbrace{\langle u_{i+1}, \sigma_{i+1} \rangle \to \langle u_{i+2}, \sigma_{i+2} \rangle \to \ldots}_{\text{steps } i < j < k} \to \underbrace{\langle u, \sigma \cdot \sigma' \cdot h \rangle}_{\text{step } k}
$$

We observe that from (1), (2), (3), the stack retains the prefix σ for all steps between i and k. From the assumption that transformations don't both push and pop the stack, and our model requirement that transformations can only match the top of the stack, this means that the forwarding for the stack at these steps does not depend on σ, and thus forall σ the subtrace starting at step i:

$$
\underbrace{\langle u, \sigma \cdot h \rangle}_{\text{step } i} \to \underbrace{\langle u_{i+1}, \sigma_{i+1} \rangle \to \langle u_{i+2}, \sigma_{i+2} \rangle \to \ldots}_{\text{steps } i < j < k} \to \underbrace{\langle u, \sigma \cdot \sigma' \cdot h \rangle}_{\text{step } k}
$$

would be the same for any such σ. For this reason, expanding out the trace from k steps to $2k - i$ steps, we observe the following continuation of the original trace:

$$
\underbrace{\langle u, \sigma \cdot h \rangle}_{\text{step } i} \to \underbrace{\langle u_{i+1}, \sigma_{i+1} \rangle \to \langle u_{i+2}, \sigma_{i+2} \rangle \to \ldots}_{\text{steps } i < j < k} \to \underbrace{\langle u, \sigma \cdot \sigma' \cdot h \rangle}_{\text{step } k} \to \ldots \to \underbrace{\langle u, \sigma \cdot \sigma' \cdot \sigma'' \cdot h \rangle}_{\text{step } 2k-i}
$$

In other words, because the forwarding between steps i and k did not depend on σ, it similarly will not depend on $(\sigma \cdot \sigma')$ for the same top of stack h between steps k and $k + (k - i) = 2k - i$ for the same loop interval. Moreover, we know that $\sigma'' = \sigma'$. This same reasoning applies inductively with the new prefix $(\sigma \cdot \sigma' \cdot \sigma')$. Thus we have an infinite loop.

Necessary ($\Rightarrow$) Let us assume there is an input ℓ that induces a loop in the network $\mathcal{N}$. We must prove that there exist $i, k \in \mathbb{N}$ and $\sigma, \sigma' \in \mathcal{H}^*$ and $h \in \mathcal{H}$ such that conditions (1), (2), and (3) hold. By way of contradiction, we assume ℓ induces a loop in $\mathcal{N}$ but that no such i, k, σ, σ', h exist to satisfy (1-3). Because the input ℓ induces a loop, we know that there is an infinite trace:

$$
\langle u_1, \sigma_1 \rangle \to \underbrace{\langle u_2, \sigma_2 \rangle}_{t_1} \to \langle u_3, \sigma_3 \rangle \to \langle u_4, \sigma_4 \rangle \to \underbrace{\langle u_5, \sigma_5 \rangle}_{t_2} \to \langle u_6, \sigma_6 \rangle \to \ldots
$$

Because the set of headers $\mathcal{H}$ comprising the hops in $\mathcal{H}^*$ is itself finite and because there is a permanent loop, there must be an infinite number of time steps $t_1, t_2, t_3, \ldots$ where the stack never goes below the size at time t_i in the future – i.e., $\forall j, \ j \geq t_i \Rightarrow |\sigma_{t_i}| \leq |\sigma_j|$. If there were no such infinite sequence, then there could not be a permanent loop since at some point t^*, the stack would have to continue to shrink forever ($\forall j_1, \ j_1 \geq t^* \Rightarrow \exists j_2, \ j_2 > j_1 \wedge |\sigma_{j_1}| < |\sigma_{j_2}|$) and would eventually become empty since stacks are finite. This would contradict the fact that there is a permanent loop since the packet would eventually be dropped when the stack becomes ε.

From the sequence of $t_1, t_2, t_3, \ldots$ and the finiteness of the topology, eventually there must eventually be a subset of t_i which we will call $t_{m_1}, t_{m_2}, t_{m_3} \ldots$ that repeat at the same node with the same top of stack:

$$\langle u_1, \sigma_1 \rangle \rightarrow \underbrace{\langle u_2, \sigma_2 \rangle}_{t_1} \rightarrow \langle u_3, \sigma_3 \rangle \rightarrow \langle u_4, \sigma_4 \rangle \rightarrow \underbrace{\langle u_5, \sigma_5 \rangle}_{t_2} \rightarrow \langle u_6, \sigma_6 \rangle \rightarrow \ldots \rightarrow \underbrace{\langle u_i, \sigma_i \rangle}_{t_{m_1}} \rightarrow \ldots \rightarrow \underbrace{\langle u_k, \sigma_k \rangle}_{t_{m_2}} \rightarrow \ldots$$

where $u_{t_{m_1}} = u_{t_{m_2}}$, and $\text{top}(\sigma_{t_{m_1}}) = \text{top}(\sigma_{t_{m_2}})$ and so on for all t_{m_i}. Because we know that at time t_{m_1} the stack $\sigma_{t_{m_1}}$ never again goes below this size, if $\sigma_{t_{m_1}} = \sigma \cdot h$, then every stack in the trace from this time on must start with σ. The earliest two times t_{m_1} and t_{m_2} capture exactly i, k in the theorem, and $\sigma_{t_{m_1}}$ captures $\sigma \cdot h$ (condition 1). The trace retains the prefix σ after time t_{m_1} (conditions 2, 3). And the nodes and top of stacks are the same at each time t_{m_i} being h (condition 2). $\qquad\square$

Enabling In-situ Programmability in Network Data Plane:
From Architecture to Language

Yong Feng[1], Zhikang Chen[1], Haoyu Song[2], Wenquan Xu[1],
Jiahao Li[1], Zijian Zhang[1], Tong Yun[1], Ying Wan[1], and Bin Liu[1]

[1]*Tsinghua University, China,* [2]*Futurewei Technologies, USA*

Abstract

In-situ programmability refers to the capability for network devices to update data plane functions and protocol processing logic at runtime without interrupting the services, driven by dynamic and interactive network operations towards autonomous networks. The existing programmable switch architecture (e.g., PISA) and programming language (e.g., P4) were designed for monolithic and static implementation, which requires a complete programming and deployment cycle for functional update, incurring long delay and service interruption. Addressing the fundamental reasons for such inflexibility, we design a new In-situ Programmable Switch Architecture (IPSA) and the corresponding design flow using rP4, a P4 language extension, as a fix. The compiler contains algorithms to support efficient resource mapping for both base design and incremental updates. To manifest the in-situ programming feasibility, we demonstrate several practical use cases on both a software switch, `ipbm`, and an FPGA-based prototype. Our experiments and analysis show that IPSA incurs moderate hardware cost which can be justified by its benefits and compensated by newer chip technologies. The in-situ programmability enabled by IPSA and rP4 advances the state of the art of programmable networks and opens a promising new design space.

1 Introduction

High-performance networking devices are usually built with hardware centered on a forwarding chipset [1–5]. The diverse network types require varied feature sets; new protocols (e.g., SRv6 [6]) and functions (e.g., INT [7]) keep emerging; meanwhile, the demand for higher throughput never relents. It becomes increasingly uneconomical or even infeasible to integrate all needed features and functions in a single chip at design time. While the future networks are expected to evolve to be autonomous with the capability of self-provisioning, self-diagnosing, and self-healing, the network operations will become more dynamic and interactive, aggravating the performance and flexibility pressure on network data plane.

We argue that the network data plane requires the in-situ programmability, which refers to the capability for network devices to update data plane functions and protocol processing logic at runtime without interrupting the services. Specifically, it ensures that (1) the on-demand and incremental part can be patched into the existing system in service without full design recompiling and reloading, (2) unused functions can be removed to preserve resource and energy, and (3) the update process has near-zero impact on network services and incurs little delay, permitting realtime interactive control loops. The need for in-situ programmability is evidenced by the following non-exhaustive list of applications:

Network slicing. A network device can be programmed to support multi-tenancy using network slicing [8, 9]. Due to the resource limitation and the application dynamics, tenants with custom policy and processing logic may be added, removed, or updated at runtime. Modifications for any tenant cannot affect the other tenants.

Network telemetry and measurement. Dynamic visibility is particularly useful to support closed control loops in autonomous networks based on realtime network conditions. However, such functions are either hard to foresee at design time or too expensive to keep permanent (e.g., sketch [10]), so it is better to make them on-demand at runtime. For example, the sketch size can be changed to get better traffic visibility as network pattern changes (e.g., DREAM [11] and SCREAM [12]); iterative debugging and query installation can be supported (e.g., Marple [13] and Path Query [14]); flows specification and associated actions can be refined and updated (e.g., Sonata [15] and ProgME [16]).

Trial on new protocols/algorithms. It was difficult to conduct live trials for new protocols/algorithms in production networks, in fear of disturbing or even disrupting network operation and incurring irrevocable damages. On the other hand, there is no better way to understand their impact and gain confidence. The dilemma can be dissolved by enabling inserting new protocols/algorithms to in-service network devices with a reliable failback procedure. Even better, a proven update can be made permanent without a network overhaul.

In-network Computing. Network devices can integrate a partial function for applications such as caching [17], aggregation [18], and coordination [19], to boost their performance and reduce system cost. Such a function can be resource-consuming but not always needed, and new functions may emerge, so it is better to make them pluggable at runtime.

Memory refactoring and repurposing. As the scarcest resource in a switch, the on-chip memory is shared by lookup tables, data cache, and packet buffers. The change of traffic pattern and network scale may raise new network optimization requirements or demand new functions, making it necessary to enlarge or shrink a table's width or depth, provision new tables, or change a search key.

State preserving for stateful functions. Conventional device updates can be destructive to the states of stateful functions stored in registers and memory tables, which need to be rebuilt from scratch or refreshed from the control plane. The detriments can be avoided if the states are preserved through hitless incremental updates.

Network data plane programmability has come a long way. The reconfigurable chips (e.g., FPGA and Network Processor) were the earlier attempts to make network devices programmable. In recent years, data plane programmability was pushed to a new height by two new developments. The packet processing and forwarding architecture was abstracted as a generic match-action pipeline (i.e., RMT-based PISA [20,21]), enabling a new type of programmable ASIC conforming to the architecture [3]; further, a high-level domain-specific language P4 [22] was developed as the chief programming language for such an architecture, which helps to accelerate the development life cycle and support design reuse and cross-platform migration. The flexibility has triggered numerous innovations, such as *in-network computing* [17, 23, 24] and *programmable network visibility* [7, 10, 25].

However, such programmability still falls short of the requirements of the aforementioned applications. The fundamental issue is that such programmability is static and limited to design time. The packet processing pipeline, once compiled and installed, cannot be changed any more during the runtime. Any new function update, no matter how minor it is, requires modifying and recompiling the complete source code, swapping in the resulting monolithic "binary", and repopulating all the tables, which inevitably introduce delay and service interruption.

Several attempts have been made from different angles to achieve higher flexibility for data-plane programmability [9, 26–29]. However, none of them can realize the desired in-situ programmability in hardware. To this end, we reason a new chip architecture other than PISA is needed, as well as the corresponding programming model. Specifically, we make the following contributions:

• We develop a new In-situ Programmable Switch Architecture (IPSA) with four key components to provide enough flexibility for in-situ programming (Sec. 2).

• We design a P4 language extension, reconfigurable P4 (rP4) (Sec. 3.1), and develop the corresponding design flow and compilers for IPSA-based device programming (Sec. 3.2); we integrate in the rP4 compiler efficient algorithms to solve the resource mapping issues raised by IPSA and incremental updates (Sec. 3.3); we detail the non-disruptive update deployment procedure (Sec. 3.4).

• We implement an IPSA-complying software behavioral model, ipbm, used as a tool to verify the rP4 compiler and test applications, similar to the role of bmv2 to P4. We also implement an FPGA-based IPSA prototype and use it to demonstrate several use cases (Sec. 4). We open source the rP4 specification, compiler, and ipbm [30]. Through experiments and analysis, we confirm that IPSA/rP4 supports non-disruptive and low-latency in-service updates, and exhibit the hardware cost and potential trade-offs (Sec. 5).

After discussing the limitations, potentials, and future work (Sec. 6), we brief the related work (Sec. 7) and conclude the paper (Sec. 8)*.

2 In-situ Programmable Switch Architecture

2.1 Motivation

To make in-situ programmability possible, it is crucial to understand why the current programmable switch architecture and programming model are incapable. We summarize the main reasons as follows:

• The packet header parser and the corresponding processing logic are decoupled. The parsing states in the standalone front parser are entangled with different pipeline stages, and a function block cannot be made self-contained and independent. Hence, an update may need to modify multiple places in a program, which is cumbersome and error-prone. Moreover, without knowing the actual processing a packet undergoes, the front parser may parse fields that the pipeline never uses, wasting parsing cycles and header vector storage.

• The pipeline stages are hardwired into a chain, on which the actual packet processing pipeline is mapped in order, resulting in several unfavorable consequences: (1) the maximum number of ingress and egress stages is fixed, limiting the design flexibility; (2) unused stages are kept in the chain, potentially increasing latency and power consumption; (3) even if each physical stage can be programmed individually, an update (e.g., inserting a stage into the pipeline) requires to reprogram all the affected stages (e.g., pushing all stages back to make room), which could be time-consuming.

• The memories for lookup tables are prorated over physical stages, implying that (1) the processing logic migration results in the associated table migration as well which increases the update delay, and (2) if the table size required exceeds

---

*This paper extends our workshop paper [31] with updates including the introduction of virtual pipeline, detailed resource mapping algorithms, non-disruptive deployment procedure, and more evaluation results.

what is provisioned in a single stage, more stages need to be combined, which reduces the effective pipeline stages.

• A pipeline-oriented P4 program can only be compiled into a monolithic "binary" file in which the individual functions are unextractable and the actual pipeline mapping is opaque to programmers, making incremental updates impossible. Some switches nominally support on-line reprogramming, suffering from considerable service interruption and packet loss.

To overcome the inflexibility of PISA and support in-situ programming, while retaining its match-action pipeline abstraction, we design a new switch architecture, IPSA, with four major architectural changes. The overview of IPSA is illustrated in Fig. 1.
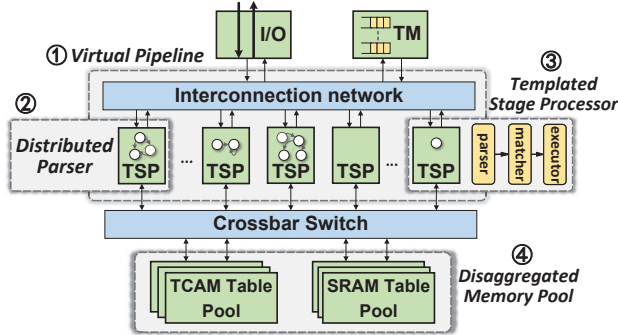
Figure 1: Overview of IPSA.

2.2 Distributed On-demand Parsing

In-situ programming implies a modular design style in which functions are self-contained. IPSA eliminates the front parser. The complete parsing graph is split into sub-graphs and distributed just in time to each pipeline stage, ensuring the self-sufficiency of each pipeline stage and avoiding unnecessary parsing. The parsing cost is amortized over active pipeline stages, making the design more scalable.

A parsing sub-graph in each stage instructs the local parsing process. Instead of a Packet Header Vector (PHV), a window of packet header bytes plus some metadata pass through the pipeline. The parsing result at each stage is recorded as {*hdr_id, hdr_offset, hdr_length*}, which is also passed to subsequent stages to avoid unnecessary re-parsing. A field in a header can be obtained using the configured {*fld_offset, fld_length*}. The design eliminates the need for deparsing at the end of a pipeline. The offset management module is responsible for adjusting the parsed header offsets in the case of header length change (e.g., MPLS label push and pop).

In the example shown in Fig. 2, the complete parser for `Ethernet`, `VLAN`, and `IPv4` is distributed into the first and the third stages. To add IPv6 support later, we can write a standalone function module which takes care of its own parsing need. There is no need to modify the other modules except for configuring the branching gateway or flow actions in the new module's direct predecessors (see Fig. 2). The distribution of
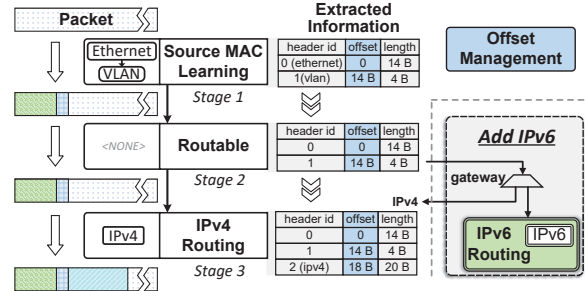
Figure 2: Distributed on-demand parsing.

a parser for a specific design is determined by the compiler. The algorithm is provided in Sec. 3.3.

2.3 Templated Stage Processor

Due to the distributed parsing, each pipeline stage processor now contains three sub-modules: a parser, a matcher, and an executor. The matcher and executor conduct the similar match-action function as in PISA.

IPSA pipeline stages are just loosely coupled, and each stage is individually programmable. By separating primitive and parameter [27, 29], each processor appears to be a parameterized container in which three abstractions are applied: (1) header fields are abstracted as *offset* and *length*; (2) flow tables are abstracted as *type*, *size*, and *key*; (3) actions are abstracted as an ordered set of primitives and their parameters. Programming a Templated Stage Processor (TSP) simply means downloading the template configurations, such as header field indicator, match type, table specification, and action, to it. TSP is a key mechanism to enable local and independent updates, allowing us to modify the function of each TSP at runtime.

2.4 Virtual Pipeline

In IPSA, the TSP interconnections are not hardwired. Instead, a reconfigurable non-blocking interconnection network (e.g., crossbar) is used. When including the packet I/O and Traffic Manager (TM) in the interconnection, we can dynamically generate arbitrary virtual pipelines in which a TSP can be allocated to any stage in either ingress or egress, regardless of its physical location, or excluded from the pipeline if unused, which can be kept in low power state to reduce heat. As long as the total number of required pipeline stages is no more than the number of TSPs, the design can be supported.
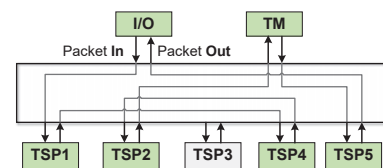
Figure 3: A virtual pipeline example.

We make a trade-off between the latency and scalability by choosing either a crossbar or a multi-stage network for TSP interconnection. Virtual pipeline maximizes the flexibility in constructing a pipeline and simplifies runtime updates. When needing to insert or remove a TSP in the pipeline, one just needs to reconfigure the interconnection network. In the example shown in Fig. 3, TSP1-TSP4-TSP2 forms the ingress pipeline, TSP5 forms the egress pipeline, and TSP3 remains in idle. The algorithm for logical stage to TSP mapping, as part of the compiler, is provided in Sec. 3.3.

2.5 Disaggregated Memory Pool

IPSA disaggregates the memory from TSPs to a shared memory pool as in dRMT [32]. A crossbar switch fabric is statically configured for each design to provide interconnection between TSPs and memory blocks. Updates on either TSPs or tables may require a reconfiguration of the crossbar. To cope with the scalability, different optimizations [32] can be used as a trade-off between flexibility and resource consumption. Specifically, we partition the TSPs and memory blocks into multiple clusters and each cluster has a crossbar for TSP-memory interconnection. In each cluster we can also apply the segment optimization [32] to further improve the scalability. Note that the clustering optimization is inapplicable to dRMT because its Run-to-Completion (RTC) processors require table replication in each cluster. The one-to-one mapping between processor and table in our architecture frees it of processor synchronization and crossbar scheduling.

Each SRAM table is mapped to some memory blocks which are not necessarily adjacent. The TCAM table virtualization technique is similar to that in RMT [20, 32]. The compiler determines memory allocation for the initial design and incremental updates. Once deployed, network operators use the APIs provided by the compiler to access the logical tables at runtime. If a logical stage is deleted, the memory blocks for its associated table are recycled.

Disaggregated memory pool allows multiple TSPs to read or write the same logical table, enabling single-pass *stateful* data-plane functions which was difficult or even impossible to realize in PISA.

3 rP4 Language and Compiler

IPSA makes local function updates possible while keeping the other incumbent functions and states intact. While IPSA paves the hardware foundation for in-situ programmability, software tools adapting to it are needed. The language should be a high-level one to ease programming, yet a paradigm shift, i.e., using a modular and stage-oriented design to replace the monolithic and pipeline-oriented design, is required. Meanwhile, we should try the best to take advantage of existing assets (e.g., P4) and avoid reinventing the wheel.

3.1 rP4 Language Overview

In IPSA, the packet processing pipeline consists of stages with each performing some parse-match-action triad. The incremental parts are inserted into the pipeline as new stages. To this end, we design a P4 language extension, rP4, dedicated to programming IPSA-based devices. The reason is multifold: P4 is familiar and supported by a mature community; we can reuse most of the existing language features; potentially we can mix rP4 code to P4 program for co-design optimization. In rP4, each *function* contains one or more *stages*, and each stage includes a *parser*, a *matcher*, and an *executor* module. The table information can be extracted from the matcher. The grammar of rP4 is given in Appendix A.

3.2 rP4 Design Flow

Illustrated in Fig. 4, the rP4 design flow comprises two parts: the base design and incremental updates upon it.
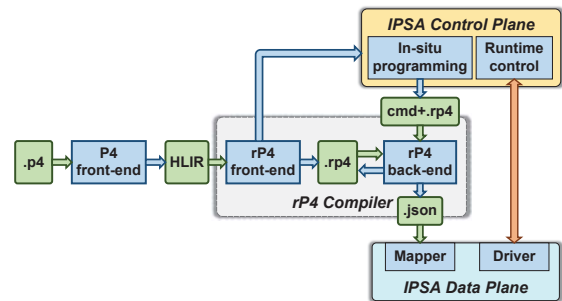
Figure 4: The complete rP4 design flow.

3.2.1 Flow for Base Design

We use P4 instead of rP4 for the original base design because P4 code is easier to write and many proven designs in P4 exist. Moreover, a design in P4 can be mapped into both PISA and IPSA-based devices, albeit the former does not support runtime incremental updates.

The rP4 front-end compiler, `rp4fc`, transforms P4 code into rP4 code. Specifically, `rp4fc` takes the HLIR, the target-independent output of `p4c`, as input, and outputs the semantically equivalent rP4 code. `rp4fc` also produces the APIs for network operators to access the tables at runtime.

To generate the final TSP and table mapping, we develop an rP4 back-end compiler, `rp4bc`. It takes rP4 code as input, analyzes the dependency of different logical stages, optimizes the predicates to merge some independent stages into a single TSP, allocates tables, and computes the best stage mapping layout. The output of `rp4bc` is the TSP templates in JSON format, which are used to configure the data-plane devices.

3.2.2 Flow for Incremental Updates

In-situ programming uses `rp4bc` as well. With the help of the rP4 base design, users gain insight into the pipeline and decide

the location for updates. To insert a new function, we write the rP4 code snippet. We then feed the commands, which stipulate the operation and location, plus the rP4 code to `rp4bc`. `rp4bc` generates two outputs: the first output is the updated base design as the reference for future updates, and the second output is the new TSP templates and switch configuration. We use another command and an rP4 function name as parameters for function deletion. Similarly, the base design is updated and new data-plane templates and configurations are generated.

3.3 Algorithms in rP4 Compiler

The rP4 compiler needs to solve two problems: the parser distribution and the mapping from logical stage to physical processor.

3.3.1 Parser Distribution and Mapping

A parser is essentially a Finite State Machine (FSM) which can be represented as a Header Parsing Graph (HPG). Fig. 5(a) shows an example of HPG in which each node represents a header. The packet processing flow is partitioned into logical stages to form a Processing Flow Graph (PFG). Each node in PFG represents a logical stage which contains a set of headers needed either for table lookup or packet processing. A PFG example is shown in Fig. 5(c).

The parser distribution problem is to determine which header(s), if available, should be parsed at each logical stage while obeying the just-in-time principle. Obviously, at each logical stage, a needed header, as well as all its predecessors in HPG, should be parsed on each path in PFG leading to the current stage. The parser distribution algorithm determines the mapping of a minimum sub-graph of HPG to each logical stage in PFG. We have two cases: the mapping for the base design and for incremental updates.

(a) HPG.

(b) Reverse mapping from header to logical stages.

(c) PFG. Below each node is the resulting distributed parser states.
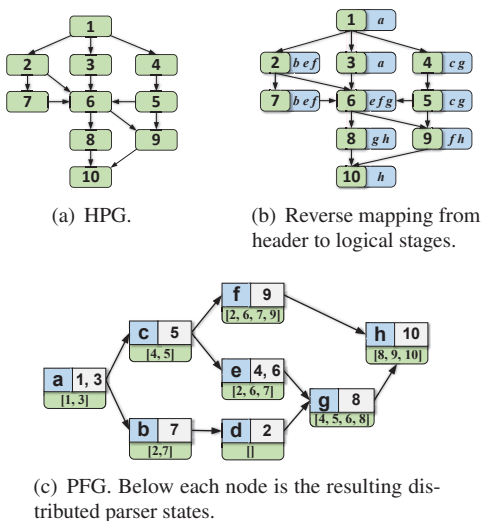
Figure 5: Mapping of distributed just-in-time parser.

Base Mapping. We construct a distributed parser for each logical stage s in the topological order of PFG. At s, for each reverse path p tracing back to the root of PFG, if a needed header i in s has been parsed, we extract a sub-graph containing i and all its predecessors in HPG which have not been parsed on p. At last, the sub-graphs for all the reverse paths are merged to generate the distributed parser for s. Fig. 5(c) shows the final mapping result for each stage if the PFG nodes is processed in the order of a-b-c-d-e-f-g-h.

To fit the internal pipeline structure of a TSP, the maximum parsing depth of a distributed parser is limited to a pre-defined value h. In case the depth H of a resulting parser exceeds h, the original logical stage is split into $\lceil H/h \rceil$ sub-stages and the parser is divided into $\lceil H/h \rceil$ sections to fit in them. Only the last sub logical stage contains the original matcher and executor. Although mapping to different TSPs, these sub-stages jointly serve as the original logical stage.

It is trivial to prove that the algorithm can guarantee the just-in-time parsing. The complexity of the algorithm is $O(V_H + E_H + V_P + E_P + V_d)$, where V_H, E_H, V_P, E_P, and V_d represent the number of vertices and edges in HPG, the number of vertices and edges in PFG, and the total number of needed headers by the logical nodes in PFG, respectively.

Incremental Update Mapping. On the basis of HPG and PFG, we can avoid rebuilding the parser mapping each time an incremental update occurs, to reduce the compiling time and update cost. However, both HPG and PFG may change as a result of the changes on protocol header, logical stage, and stage transition. To solve the problem, we establish a reverse mapping from HPG nodes to PFG nodes. Each HPG node i is associated with a set of logical stages in which the header i is parsed. The result for our example is shown in Fig. 5(b).

In PFG, the parser change on s does not influence its predecessors. If a removed header i in s may cause another header j in some predecessor stage s' to become redundant, it means j is not needed in s' in the first place. The just-in-time parsing makes this case impossible. If a new header i is added to s, s is solely responsible for parsing all i's predecessors in HPG that are not parsed yet on all the paths leading to s in PFG. Therefore, we have the following procedure for two cases of HPG change. (1) A header i insertion or deletion in s: find all the direct successors of i in HPG and get their corresponding logical stages from the reverse mapping. Update the parsers in s and these logical stages as well as their successors in topological order of PFG. (2) A topology change in HPG: get the corresponding logical stages from the reverse mapping for all the influenced headers and update the parsers in these stages and their successors in the topological order of PFG. During the update, if all the direct predecessors of s' do not change their parsers, then s' does not need to change its parser either, so the update process can stop earlier.

Similarly, for a change in PFG, we have the following two cases. (1) A logical stage s insertion or deletion: update the parsers in all s's successors in topological order of PFG. (2)

A topology change in PFG: find all the influenced stages and their successors, and update the parsers in these stages in topological order of PFG. Although the time complexity is the same as the base mapping, in practice the incremental update mapping is much faster.

3.3.2 Logical-to-physical Topology Mapping

Unlike the logical-to-physical pipeline mapping problem in PISA [20, 33], the PFG-to-TSP mapping in IPSA faces different freedom and constraint due to virtual pipeline and disaggregated memory cluster. The high level goal is to minimize the number of required TSPs and maximize the potential to support incremental updates.

Assume there are m TSP clusters, $\mathbb{C} = (C_1, C_2, ..., C_m)$, and each cluster i has n TSPs $P_i = (p_{i,1}, p_{i,2}, ..., p_{i,n})$ sharing s SRAM blocks and t TCAM blocks.

Base Mapping. Let $P(v)$ denote the TSP to which the logical stage represented by node v in PFG is mapped and $V(p)$ denote the set of independent logical stages mapping to the TSP p. We model the mapping from PFG to TSP as an ILP problem with the following constraints and objectives:

Constraint 1: Successor Exclusion. Any two logical stages cannot be mapped to the same TSP if they are on the same path in PFG. That is,

$$P(v_i) \neq P(v_j), \ if \ v_i \succ v_j \ or \ v_j \succ v_i \qquad (1)$$

in which "$\succ$" denotes the successor relationship.

Constraint 2: Path Order. The active TSPs form a pipeline on which the logical stages on the same path in PFG must follow the pipeline order. That is,

$$\forall \ v_i, v_j \in V, \ if \ v_i \succ v_j \Rightarrow P(v_i) \succ P(v_j) \qquad (2)$$

Constraint 3: TSP Capacity. The number of parallel logical stages that can be mapped to a single TSP is limited to a predefined value, K, depending on the TSP resource. That is,

$$\forall \ p \in P, \ |V(p)| \leq K \qquad (3)$$

Constraint 4: Flow Table. The total number of memory blocks required by the logical stages mapped to the TSPs in a cluster should not exceed the available resource. That is,

$$\forall \ C_i, \ \sum_{1 \leq j \leq n} s(p_{i,j}) \leq s, \ \sum_{1 \leq j \leq n} t(p_{i,j}) \leq t \qquad (4)$$

in which $s(p)$ and $t(p)$ denote the number of SRAM and TCAM blocks required by the TSP p, respectively.

Objective 1: To save more TSPs for future updates, the number of active TSPs should be minimized by mapping independent logical stages to the same TSP. Let $a(p)$ be 1 if p is active and otherwise be 0. The objective is therefore,

$$\min \sum_{1 \leq i \leq m, 1 \leq j \leq n} a(p_{i,j}) \qquad (5)$$

Objective 2: The initial mapping should satisfy the processor and memory requirements with as few clusters as possible, so as to concentrate the unused resources in some clusters to make logical stage and table allocation for future updates easier. Approximately, the objective is expressed as,

$$\max \sum_{C_i \in \mathbb{C}} m_i^2 \sum_{1 \leq j \leq n} (\frac{K - u_{i,j}}{K})^3 \qquad (6)$$

in which m_i is the ratio of free memory blocks in cluster C_i, and $u_{i,j}$ is the number of used stage resources in $p_{i,j}$. The formula favors more free processors.

We use the open-source ILP solver YALMIP [34] to solve the problem. For the example in Fig. 6(a), the base mapping result is shown in Fig. 6(b), and the virtual pipeline is $(a) \rightarrow (c) \rightarrow (b,f) \rightarrow (d,e) \rightarrow (g) \rightarrow (h)$.

Incremental Update Mapping. To make incremental changes for each runtime update (e.g., insertion or deletion of a function), we use a *greedy mapping* algorithm other than ILP to obtain a local optimal solution, because ILP is not only slower but also possible to significantly change the mapping result which requires excessive stage and table migrations.

Greedy Mapping. We maintain a profile for each cluster to record its free SRAM blocks, TCAM blocks, and the usage of TSPs (Fig. 6(b)). The logical stage insertion performs the following steps: first, exclude the clusters without enough free memory blocks required by the new stage; second, check whether any processor in the remaining clusters can accommodate the new stage under the constraints (1), (2), and (3); third, in the feasible clusters, choose the one based on the objective (6). In Fig. 6(a), a new stage **i** which needs 2 SRAM blocks is inserted. $p_{3,2}$ is selected as the greedy mapping result shown in Fig. 6(c).

3.4 Non-disruptive Update Deployment

After update compiling, the update deployment handles the device configuration. Since an update may need to insert or delete multiple logical stages on multiple TSPs, the device configuration involves multiple tasks: initialize the TSP templates and logical tables, reconfigure the TSP-memory crossbar and the virtual pipeline, and modify the transitional logic of the affected predecessor stages. The update deployment needs to meet three requirements. (1) *Consistency*: any packet in pipeline must be processed either before or after an update takes effect; (2) *Non-disruption*: the deployment process should not cause service interruption or packet drop; (3) *Low latency*: the time taken should be minimized.

The deployment procedure we use is named Big Bubble Update (BBU). BBU can make an update take effect within a fixed time window at the cost of a small buffer in front of the processing pipeline. As illustrated in Fig. 7, any update can be decomposed into a set of three basic operations:

MOD. When needing to modify logical stages in TSP2 (and any other TSPs after TSP2), TSP1 is first stopped from

(a) PFG with SRAM (S)/TCAM (T) requirements for each stage. 'd' and 'w' denote the depth and width of a memory block, respectively.

(b) Base mapping result. There are 4 clusters with each having 2 TSPs, 16 SRAM blocks, and 4 TCAM blocks.

(c) The result mapping after inserting stage **i**.
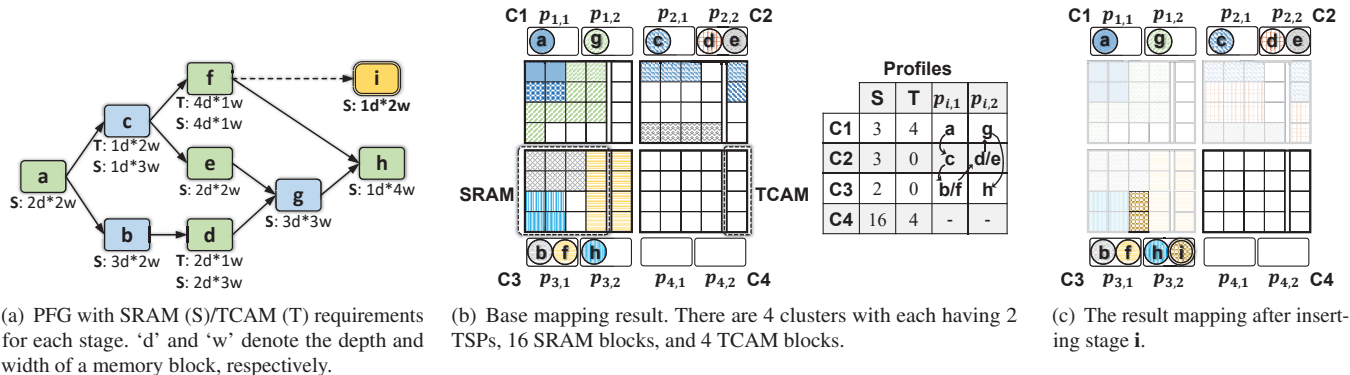
Figure 6: The base and runtime update mapping from logical stages to physical processors.

moving packets to TSP2 to drain TSP2 in time T. After that, d more clock cycles are used to configure TSP2. Then the packet flowing resumes. The other TSPs that need to be modified will take turn when the created bubble arrives.

DEL. The gateway in a TSP determines in which following TSP and logical stage a packet should be processed. When needing to deleting a logical stage s in TSP3, the preceding TSPs need to modify their gateways if their direct target is s.

INS. It is much easier to insert a new TSP with new logical stages into the pipeline. There is no need to halt any part of the pipeline during the d clock cycles used for new TSP configuration. Both DEL and INS just need one clock cycle to reconfigure the pipeline interconnection as the last step.
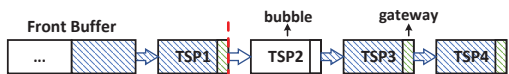
Figure 7: BBU example. When TSP1 is halted, new arrival packets are accumulated in the front buffer.

BBU guarantees the update consistency (i.e., any packet cannot be partially processed by an updated function). A MOD update takes effect after at most $(T+d)$ clock cycles, and DEL and INS updates take much shorter time, meaning that an update can be performed as soon as there is enough space for $(T+d)$ packets in the front buffer. A complex example in Fig. 8 shows that multiple updates can be achieved with one big bubble as well.

4 Implementation and Use Case Demo

To verify the architecture and programming flow, we build both software and hardware IPSA prototypes, on which several use cases are demonstrated.

4.1 IPSA Prototypes

Software Switch: We implement a behavioral model, `ipbm`, on Ubuntu 20.04 LTS as a reference software switch con-
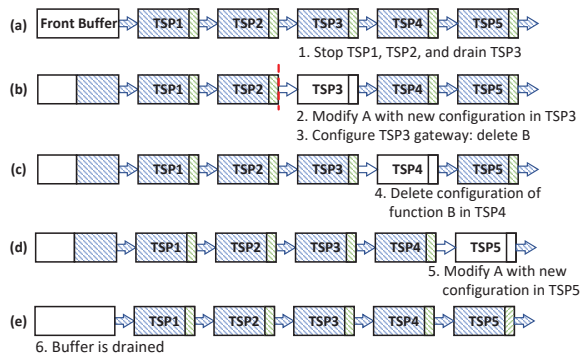
Figure 8: Function A resides in TSP3 and TSP5; Function B resides in TSP4. To modify A and delete B, the updates are performed in order when the target TSP is in the big bubble.

forming to IPSA. `ipbm` takes 8,361 lines of C++ code. `ipbm` consists of four modules: the Communication Module (CM) supports OS kernel bypass and direct packet I/O; the Pipeline Module (PM) simulates the TSPs; the Control Channel Module (CCM) communicates with the controller for runtime configuration; the Storage Module (SM) realizes the disaggregated memory pool.

Hardware Switch: We build a hardware prototype on a Xilinx Alveo U280 accelerator card. The Xilinx 16nm Ultra-Scale+ FPGA contains 8GB of HBM2 memory with 460G/s bandwidth [35]. We implement both IPSA (2,366 lines of Scala code) and PISA (1,942 lines). Each prototype contains 12 physical processors (K=2). The TM is omitted for simplicity. Each IPSA TSP supports a 192-byte packet window, 64-byte metadata, and a 4-level pipelined parser. The TSPs are partitioned into 3 clusters, each with 64 256×64b memory blocks. The maximum bus-width for memory access is limited to 256-bit (i.e., four memory ports) for both prototypes. Each executor contains four primitives which are sufficient to our use cases. We implement both memory blocks and virtual pipeline interconnections with a 12×12 full crossbar. The PISA prototype realizes a 256-byte PHV which comprises 32x 8-bit, 48x 16-bit, and 32x 32-bit containers. Each
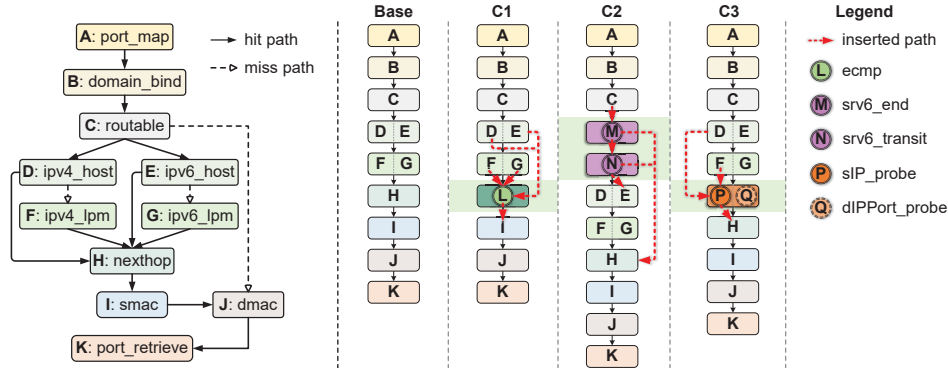
Figure 9: The packet processing flow and TSP pipeline mapping for the use cases.

processor in PISA can access 16 memory blocks.

Compiler and Controller: rp4c is implemented with 3,772 lines of C++ code. The controller is used for runtime configuration and in-situ programming. We implement a simple command-line interface in C++, allowing users to load or offload on-demand protocols and functions at runtime.

4.2 Base Design Compiling Results

We compile several open source P4 projects [17, 36–39] for ipbm and bmv2. Table. 1 shows the number of logical stages (*LS*) and the number of logical pipeline levels (*LPL*) on ipbm. bmv2 produces the same LPL results. The table also shows the average depth of the distributed parsers (*ADP*) and the percentage of the distributed parsers whose depth is under 5 (*U5*), confirming 4 is a good trade-off for the supported parser depth in a TSP.

	LS	*LPL*	*ADP*	*U5* (%)
switch.p4 [36]	130	31	0.28	100
DC.p4 [37]	38	19	0.45	97.37
ONTAS [38]	22	8	0.36	100
P4SRv6 [39]	17	5	0.53	100
NetCache [17]	96	14	0.21	100

Table 1: Design compiling results.

4.3 In-situ Programming Use Cases

To fit in our hardware prototype, the base design, as shown in Fig. 9, is extracted from switch.p4, which includes L2 switching with IP subnet-based VLAN and L3 forwarding based on IPv4/IPv6. The workflow is as follows: (1) get interface index via port mapping table (**A**), (2) bind the Bridge Domain (BD) and the Virtual Routing Forwarding (VRF) table (**B**), (3) determine L2 or L3 forwarding (**C**), (4) derive the egress interface index via BD and dMAC (**J**), (5) process IPv4/v6 header and get the next-hop (**D, E, F, G**), (6) update BD and dMAC via nexthop (**H**), (7) update sMAC via updated BD (**I**), (8) get the egress port via egress interface index (**K**). As shown in Fig. 9, the resulting PFG contains 11 logical stages

mapping to 9 TSPs. To showcase the in-situ programming capability, we select three representative applications which introduce new functions or protocols to the switch at runtime.

C1: Equal-Cost Multi-Path Routing (ECMP). While there are multiple network load balancing algorithms, we choose ECMP [40, 41] as an example to augment the base design. After the FIB lookup, the function chooses a forwarding link based on the next-hop and flow ID hashing. ECMP does not introduce new protocols, but two new tables and processing logic. The rP4 code for the ECMP function is shown in Fig. 10(a). ECMP applies for both IPv4 and IPv6. Since they are independent, only one physical stage is needed. The function also covers and therefore replaces the stage **H**. To insert the ECMP function into the original switch, we first compile the function code and the associate configuration script (Fig. 10(b)) into template parameters and required topology modifications, and then apply the configurations on the device. In this case, users need to link IPv4 forwarding and IPv6 forwarding with ecmp. The links from and to the original nexthop are removed through 'delete_link' command to eliminate the old function from the pipeline.

C2: IPv6 Segment Routing (SRv6). SRv6 [42] is an IPv6-based source routing protocol. It uses a new IPv6 extension header (i.e., SRH) to carry the forwarding path information [6, 43]. The SRv6 function has two sequential logical stages, srv6_end and srv6_transit, for SR end-point and transit-node processing, respectively. A packet first goes through the srv6_end stage. If the packet's SID matches the local SIDs of the switch, the end-point function is executed; otherwise, the transit-node function in the srv6_transit stage is executed, which could insert an SRH to the packet or simply forward the packet. In this case, the script for loading the function needs to link the new header into the original header list (Fig. 10(c)). Since the switch should still support pure L3 forwarding, the linkage between routable and ipvx is reserved. After rp4bc compiling and configuration downloading, the target is renewed with SRv6 support.

C3: Dynamic Flow Probe. To realize dynamic network measurement [11, 14], we insert an event-triggered probe at runtime and later the probe can be updated to change the object

```
1  /***** table definition: ecmp_ipv4, ecmp_ipv6 *****/
2  field_list ecmp_v4 {ipv4.src_addr; ipv4.dst_addr; ipv4.ip_proto}
3  ⋯ // action profile definition same as P4
4  table ecmp_ipv4 {
5      key = { meta.nexthop: exact; }
6      action_profile: ecmp_v4_profile; // do hash on ecmp_v4
7      size = 256;
8  }
9  table ecmp_ipv6 { ⋯ } // similar with ecmp_ipv4
10 /***** stage/function definition: ecmp function *****/
11 stage ecmp { // parser => matcher => executor
12     parser { ipv4, ipv6 }; // define headers ecmp needs
13     matcher {
14         if(ipv4.isValid()) ecmp_ipv4.apply();
15         else if(ipv6.isValid()) ecmp_ipv6.apply();
16         else;
17     };
18     executor { // execute actions according to matching result
19         1: set_bd_dmac;
20         default: NoAction;
21     }
22 }
23 /***** action definition: set egress bridge and dmac *****/
24 action set_bd_dmac(bit<16> bd, bit<48> dmac) {
25     meta.routed = true; // table hit, the packet can be routed
26     meta.bd = bd;
27     ethernet.dst_addr = dmac;
28 }
```

(a) The rP4 code for the ECMP function.

```
1  load ecmp.rp4 --func_name ecmp
2  del_link ipv4_host nexthop
3  del_link ipv4_lpm nexthop
4  add_link ipv4_host ecmp
5  add_link ipv4_lpm ecmp
6  del_link nexthop smac
7  add_link ecmp smac
8  ⋯ // omit IPv6 topology change similar with IPv4
```

(b) The script for loading ECMP to `rp4bc`.

```
1  load srv6.rp4 --func_name srv6
2  ⋯ // omit stage topology change
3  link_header --pre IPv6 --next SRH --tag 43
4  link_header --pre SRH --next IPv6 --tag 41 // inner IPv6
5  link_header --pre SRH --next IPv4 --tag 4 // inner IPv4
```

(c) The script for loading SRv6 to `rp4bc`.

Figure 10: Code and script for runtime programming.

and trigger criteria. Specifically, we implement a heavy hitter detector based on SIP. Once a flow's traffic exceeds a threshold, the probe is triggered and user can apply pre-defined ACL or QoS rules to the flow. After a while, we switch the monitoring focus by using {DIP, DPORT} as the key, which requires policy update and table refactoring. The TSP mapping result is shown in Fig. 9. Since the probe works for IPv4, a link from IPv4 forwarding to the probe is added.

Due to space limitations, we omit the case for function/protocol removal, which is usually simpler than insertion.

5 Evaluation

First, we study the hardware cost for IPSA-based chips based on the FPGA prototype, theory analysis, and empirical evidence from previous study [20, 32]. Second, we conduct

experiments on the prototypes for IPSA and PISA using the aforementioned use cases to examine the performance such as forwarding throughput and latency, compiling time and configuration time for incremental updates, and power consumption. Due to the lack of real ASIC implementations, for some aspects, we can only gain the relative performance by comparing the IPSA and PISA prototypes.

5.1 Hardware Cost Analysis

We first analyze the cost of each key component and then provide an overall evaluation.

TSP. Table. 2 compares the FPGA resource (LUT and FF) consumption for a processor and shows that an IPSA TSP consumes 0.581% fewer LUTs and 0.847% more FFs than a PISA processor. The higher register consumption of IPSA is due to the need for template parameter storage.

Unit	LUT (%)		FF (%)	
	PISA	IPSA	PISA	IPSA
Parser	-	1.256%	-	0.684%
Matcher	4.131%	0.697%	0.295%	0.243%
Executor		1.597%		0.215%
Total	4.131%	3.550%	0.295%	1.142%

Table 2: Processor resource in FPGA prototypes.

Interconnection network for virtual pipeline. The number of TSPs determines the scale of the interconnection network. Different types of interconnection networks have different scalability and latency trade-offs. For N TSPs, we consider four types of non-blocking networks, i.e., Crossbar, Clos [44], Benes [45], and Batcher Banyan (BB) [46–48], which have characteristics in Table. 3.

Type	Cross-points	Latency (cycles)
Crossbar	N^2	1
Clos	$2Nc + N^2/c$	3
Benes	$4(N\log_2 N - N/2)$	$2\log_2 N - 1$
BB	$N\log_2^2 N$	$\log_2 N(1 + \log_2 N)/2$

Table 3: Cross-point comparison. c is the number of sub-switches in the second stage of Clos.

Clos is a good compromise between resource and latency. It is easy to derive that when $c = \sqrt{N/2}$, the minimum number of cross-points is achieved. For 32 TSPs, Clos can save 50% cross-points of Crossbar when $c = 4$. The network is composed of sixteen 4×4 crossbars and four 8×8 crossbars. Comparing to the internal latency of a TSP ($21 \sim 29$ clock cycles), the Clos network only adds three more cycles per stage. Based on the same assumption as in dRMT [32] (e.g., $200mm^2$ die size on 28nm technology for the entire chip), for an ASIC implementation, the die size of the Clos with 4Kb data bus width would be about $4.173mm^2$.

IPSA prototype implements a 12×12 crossbar for TSP interconnection. The data bus comprises 192-byte headers, 64-

byte metadata, 32-byte extracted information, 4-byte parsing states, and 6-bit configuration information (2,090 bits in total). The crossbar consumes 17.48% LUTs and 1.02% FFs of the FPGA, which account for 27.93% LUTs and 6.92% FFs consumed by all the TSPs, respectively.

Crossbar between TSPs and memory. Crossbar is used for memory access mainly due to the latency concern. dRMT uses a one-to-many segment crossbar to trade off the flexibility for scalability [32]. Equivalent to the configuration of RMT [20], it has 32 collections of memory blocks and 32 processors with each owning eight 80-bit memory access ports. Each matching key of a processor connects to a port in each memory collection, so the number of cross-points is $32 \times 8 \times 32 = 2^{13}$.

For IPSA, if we partition the 32 TSPs and memory collections into 8 clusters, and allow each TSP to connect to all ports in a memory collection, the number of cross-points would be $4 \times 8 \times (8 \times 4) \times 8 = 2^{13}$ as well. As a generalization, if we have M TSPs and M memory collections which are partitioned into m clusters, and each TSP connects to k memory ports, the number of cross-points is $M^2 k^2 / m$. When $M = 32$ and $k = 8$, by varying m, we get results in Table 4, in which the flexibility indicates the number of memory collections each TSP can access.

Type	m	Cross-points	Flexibility
IPSA	2	2^{15}	16
	4	2^{14}	8
	8	2^{13}	4
	16	2^{12}	2
dRMT	1	2^{13}	4

Table 4: Crossbar cross-point and flexibility comparison.

IPSA offers a wide design space for crossbar configuration. Higher memory flexibility (e.g., the capability to support large tables and the freedom for table mappings) can be gained with larger crossbar area. The crossbar in dRMT can be considered as a special case for IPSA. However, due to the lack of clustering, each of dRMT's processors needs to reach all the 32 memory collections, which increases the chip wiring latency and complexity. In contrast, to achieve the same number of cross-points, IPSA allows 8 clusters, and each TSP only needs to reach four memory collections.

Our IPSA prototype splits 12 processors into 3 clusters, resulting in 768 cross-points. The crossbar consumes 3.08% LUTs and 0.01% FFs of the FPGA, which account for 4.92% LUTs and 0.07% FFs of the IPSA prototype, respectively. Similarly, for an ASIC implementation with 32 TSPs, 8 clusters, eight 80-bit matching width, and return data containing eight 10-bit action pointers and 96-bit action data segments, the chip area of the crossbar is about $1.728mm^2$, similar to the result of dRMT.

Put everything together. The consumption of SRAM and TCAM is the same for IPSA and PISA, so the comparison

is omitted. Front parser and deparser are unique for PISA. The overall comparison of the two prototypes is shown in Table 5. The IPSA prototype consumes 12.09% more LUTs and 9.69% more FFs than the PISA prototype.

Protoype	PISA		IPSA	
Resource	LUT	FF	LUT	FF
Parsers/Deparsers	0.94%	1.54%	-	-
Processors	49.55%	3.52%	42.02%	13.72%
Crossbar	-	-	3.08%	0.01%
Inter-Network	-	-	17.48%	1.02%
Total	50.49%	5.06%	62.58%	14.75%

Table 5: FPGA resource for PISA and IPSA prototypes.

5.2 Experiment Settings
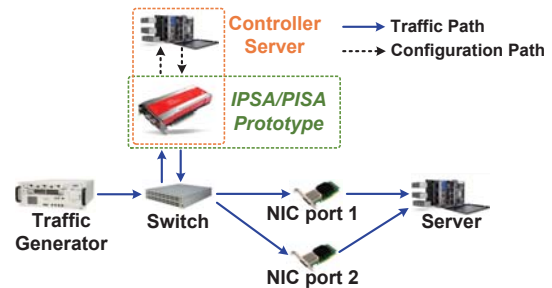
Figure 11: Testbed configuration.

Testbed. As shown in Fig. 11, the testbed is composed of the FPGA-based switch prototype, a server as the controller for switch configuration and control, a Spirent SPT-N4U-220 traffic generator [49] to generate test traffic, a server equipped with a Mellanox ConnectX-5 dual-port 100G NIC, and an Edgecore Wedge100BF-32X [50] switch to connect the devices. The FPGA has two 100Gbps QSFP28 Ethernet ports for data path traffic and one PCIe 4.0 interface supporting up to 16GT/s to the controller server. The Spirent SPT-N4U can generate up to 400Gbps packet trace with the accuracy of 10ns per frame. The traffic generator sends packets to the server through the FPGA. Because the FPGA has only one egress port, the Edgecore switch is used to split the traffic to the two NIC ports in order to demonstrate the ECMP function. The Edgecore switch is also programmed to timestamp the packets to and from the FPGA for latency measurement.

Packet Trace. Based on the use cases in Sec. 4.3, three types of packets shown in Table 6 are generated to test the prototypes. All the packets are 192-byte long with different numbers of padding bytes as payload. Each type of packet amounts to one third of the generated traffic.

5.3 Performance Evaluation

5.3.1 Switch Throughput and Latency

Throughput. With Vivado Design Suite [51], the synthesized clock frequency for IPSA is 110.45MHz and for PISA is

Header format	Header length (bytes)
`Ethernet-VLAN-IPv4-UDP`	46
`Ethernet-VLAN-IPv6-UDP`	66
`Ethernet-VLAN-IPv6-SRv6-UDP`	112

Table 6: Test packet types.

153.30MHz. The lower clock frequency for IPSA is due to the wiring complexity of the interconnection networks, which can be improved by the ASIC implementation. Benefited from the full pipelined design, theoretically, the IPSA and PISA prototypes can support a throughput of 169.65Gbps and 235.47Gbps, respectively. However, because the FPGA only has a 100Gbps interface, the peak throughput of the two prototypes is limited to 100Gbps.

Latency. We measure the forwarding pipeline latency based on the ingress and egress timestamps on packets. The results are shown in Fig. 12(a). The longer latency of the IPSA prototype is due to field and flow table profile fetching, match key assembly, and primitive loading. The gap can be mitigated and even reversed if the number of processors is large and the number of active processors is relatively small.

(a) Forwarding latency. (b) IPSA average pipeline latency.
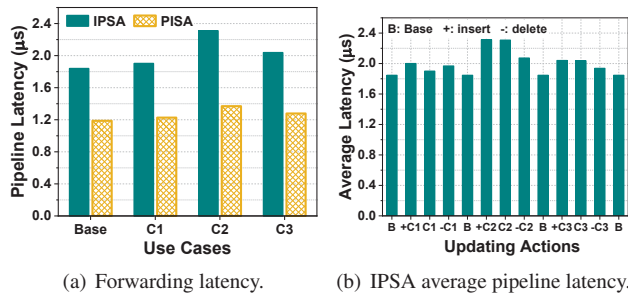
Figure 12: Pipeline forwarding latency.

5.3.2 Incremental Update Deployment Delay

In addition to the rP4 design flow, we also implement the use cases in P4 design flow for comparison. Each time the updated source code is compiled by `p4c` and a PISA-based back-end compiler, and the FPGA prototype is loaded with the updated design.

The update process of PISA consists of two phases: compiling the updated code and reflashing the device. The latter causes pipeline interruption. In contrast, IPSA decomposes the second phase into two parts: configuration loading and update executing. Only the second part halts the pipeline. We use t_C, t_L and t_H to denote code compiling time, configuration loading time, and pipeline halting time, respectively. The update performance of PISA and IPSA is shown in Fig. 13. Similar comparison between `bmv2` [52] and `ipbm` is also included in terms of compiling time and halting time.

As shown in Fig. 13(a), since IPSA only compiles the updated code segment, it takes much shorter time than PISA. Fig. 13(b) shows that t_L of IPSA is much shorter than t_C. Fig. 13(c) exhibits that IPSA's pipeline halting time is only

(a) Compiling time t_C. (b) Loading time t_L of IPSA.

(c) Pipeline halting time t_H. (d) Overall update time.
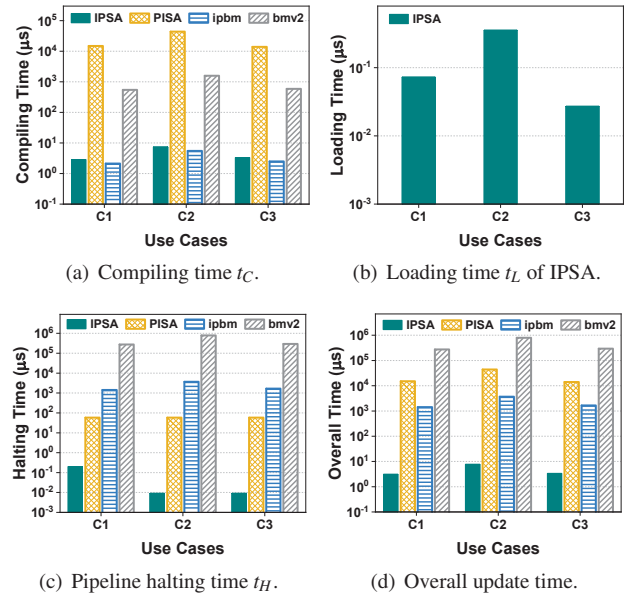
Figure 13: Update performance.

0.34% of PISA's, allowing a small front buffer of 22 packets. Fig. 13(d) sums t_C, t_L, and t_H as the overall update time, showing that IPSA has much better update performance than PISA. The time comparisons between `ipbm` and `bmv2` in Fig. 13 lead to the same conclusion.

Fig. 12(b) shows the average pipeline latency before, during, and after each update in IPSA. C1, C2, and C3 are inserted and removed sequentially. While no packet drop is observed, the latency fluctuation is also small, revealing that the update deployment process of IPSA has negligible impact on packet forwarding. In contrast, any update in PISA needs to take the device offline and repopulate the tables, incurring longer latency and higher impact on packet forwarding.

5.3.3 Power Consumption

As a side benefit, the virtual pipeline in IPSA helps reduce the chip power consumption. We extend the number of processors to 32 and infer the power consumption of IPSA and PISA with different number of active processors using the Vivado Design Suite. We assume that the unused TSPs in IPSA are put in idle state while all the processors in PISA are active in the pipeline. As shown in Fig. 14, IPSA consumes less power when the number of active processors is smaller than 18. Since the extra interconnection networks in IPSA are both passively configured, we expect the ASIC implementation can achieve even better power efficiency.

6 Discussion and Future Work

Whenever possible we try to reuse the fruition of P4 and PISA in our design unless the issue is unique to our architecture.
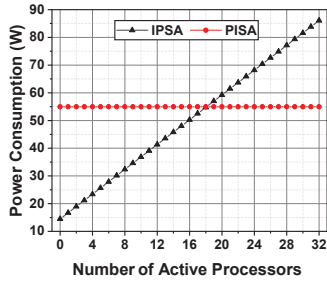
Figure 14: Power consumption in terms of active processors.

Numerous design details omitted due to space limit in this paper are documented on the open source website.

In addition to the concurrency-based processor reduction [33], the resource mapping algorithm for updates can also be enhanced. To accommodate a new stage with memory requirement exceeding the free memory in any cluster, it is possible to relocate some existing stages to different clusters or split the new stage into multiple clusters.

The resource penalty for supporting IPSA can be offset by its unique properties and compensated by newer chip technologies: (1) A typical forwarding chip is usually built with multiple parallel pipelines. While PISA requires table replications in each pipeline, which reduces the effective memory resource, IPSA allows multiple pipelines to share a single copy of each table if multi-port memory blocks are provided. (2) In PISA, a big flow table requires combining multiple physical processors, reducing the effective pipeline stages. In IPSA, a logical stage can always map into a single TSP as long as its memory requirement can be satisfied by a cluster. (3) Since only active TSPs are kept in the pipeline in IPSA, the pipeline latency can be reduced, which offsets the extra latency introduced by the interconnection networks. (4) In IPSA, the statically configured interconnections for virtual pipeline and memory are more power-efficient than the dynamic switching network in dRMT. (5) The disaggregated architecture of IPSA also allows homogeneous components to be built on separate silicon chips and integrated with the 3D-IC technology [53–55], effectively expanding the available resource and reducing the memory access latency. It is conceivable to have a three layer chip architecture composed of processor, interconnection fabric, and memory. The detailed chip design and evaluation will be attended as future work.

The interconnection network allows the processors to be organized into a directed graph instead of a pipeline, which brings new design possibilities and challenges for parallel processing, deserving further research. On the other hand, while the full interconnection is resource intensive, we can explore the design space leaning to better resource efficiency but less flexibility as a trade-off (e.g., partial interconnection, blocking network, or bypassable pipeline stages).

It is also interesting to explore the possibility to automate the rP4 code generation by comparing the difference between the old and updated P4 programs. A GUI-based development environment would help visualize the pipeline and ease the programming process.

7 Related Work

dRMT [32] also decouples processors and memory, demonstrating the feasibility of resource pooling and crossbar-based interconnection; however, the RTC mode of processors excludes the possibility of incremental updates. POF [56] allows runtime table and function insertion into data-plane devices, but only applies on software-reprogrammable network processors rather than ASIC. IPSA adopts the similar approach as in POF to support distributed parsing. Some software switches (e.g., VPP [57]) support runtime updates as well but the techniques cannot be ported to hardware. daPIPE [58] allows users to integrate custom functions into the preexisting data-plane program, but still requires recompiling the integrated program. Mantis [59] supports predefined malleable values, fields, and tables whose semantics can be changed during runtime for reactive programming. While this is a step towards runtime behavior changing, the flexibility is limited and fine-grained, and the behavior must be predefined at design time. Hyper4 [9] virtualizes the data plane to adapt to various forwarding applications. Newton [29] supports a query template for dynamic telemetry, which is hard to extend. Some other works [8,27,60] virtualize network functions and match tables, but cannot support runtime data-plane programming. Limited to FPGA, Partial Reconfiguration (PR) [61] allows users to reconfigure pre-allocated regions at runtime. However, the performance and scalability issues make FPGA unsuitable for core switching chip, and the flexibility and deployment delay of PR still cannot match that of IPSA. Designed for smart NIC, PANIC [62] also uses a switching fabric for flexible compute unit interconnection, but a scheduler is needed to schedule the service chain for each packet.

8 Conclusion

IPSA and rP4 open a new design space for network programmability, enabling new applications in the era of autonomous networks. Our implementation and evaluation have demonstrated the feasibility and benefits of the new chip architecture and programming model. We open source the rP4 specification and compiler along with ipbm, with the expectation that our work can inspire a new breed of switch ASICs, engage the community to advance the language support, and help gestate novel in-situ programmable applications.

References

[1] BCM56960 Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56960-series.

[2] Innovium TERALYNX. https://www.innovium.com/teralynx.

[3] Intel Tofino 2. https://ark.intel.com/content/www/us/en/ark/products/210608/intel-tofino-2.html.

[4] NVIDIA Mellanox SPECTRUM-2. https://www.mellanox.com/files/doc-2020/pb-spectrum-2.pdf.

[5] Trident3-X7 / BCM56870 series. https://www.broadcom.cn/products/ethernet-connectivity/switching/strataxgs/bcm56870-series.

[6] Clarence Filsfils, Darren Dukes, Stefano Previdi, John Leddy, Satoru Matsushima, and Daniel Voyer. IPv6 Segment Routing Header (SRH). RFC 8754, March 2020.

[7] In-band Network Telemetry (INT) Dataplane Specification. https://github.com/p4lang/p4-applications/blob/master/docs/INT_v2_1.pdf.

[8] Peng Zheng, Theophilus Benson, and Chengchen Hu. P4visor: Lightweight virtualization and composition primitives for building and testing modular programs. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, pages 98–111, 2018.

[9] David Hancock and Jacobus Van der Merwe. Hyper4: Using p4 to virtualize the programmable data plane. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pages 35–49, 2016.

[10] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 561–575, 2018.

[11] M. Moshref, Minlan Yu, R. Govindan, and Amin Vahdat. DREAM: Dynamic Resource Allocation for Software-Defined Measurement. In *SIGCOMM*, 2014.

[12] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. SCREAM: Sketch Resource Allocation for Software-Defined Measurement. In *Proceedings*

of the 11th ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT), 2015.

[13] S. Narayana, Anirudh Sivaraman, V. Nathan, Prateesh Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and Changhoon Kim. Language-Directed Hardware Design for Network Performance Monitoring. *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2017.

[14] Srinivas Narayana, Mina Tahmasbi, Jennifer Rexford, and David Walker. Compiling Path Queries. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, 2016.

[15] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *ACM SIGCOMM*, 2018.

[16] Lihua Yuan, Chen-Nee Chuah, and Prasant Mohapatra. ProgME: Towards Programmable Network MEasurement. *IEEE/ACM Transactions on Networking*, 19(1):115–128, 2011.

[17] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles, Shanghai, China*, 2017.

[18] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. In-network computation is a dumb idea whose time has come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks (HotNets)*, 2017.

[19] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2018.

[20] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. *ACM SIGCOMM Computer Communication Review*, 43(4):99–110, 2013.

[21] N McKeown. Protocol-independent switch architecture (PISA). https://forum.stanford.edu/events/2016/slides/plenary/Nick.pdf.

[22] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming Protocol-Independent Packet Processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[23] Zhaoqi Xiong and Noa Zilberman. Do switches dream of machine learning? toward in-network classification. In *Proceedings of the 18th ACM workshop on hot topics in networks*, pages 25–33, 2019.

[24] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael M Swift. ATP: In-network aggregation for multi-tenant learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021.

[25] Tian Pan, Enge Song, Zizheng Bian, Xingchen Lin, Xiaoyu Peng, Jiao Zhang, Tao Huang, Bin Liu, and Yunjie Liu. INT-Path: Towards Optimal Path Planning for In-Band Network-wide Telemetry. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 487–495. IEEE, 2019.

[26] Johannes Krude, Jaco Hofmann, Matthias Eichholz, Klaus Wehrle, Andreas Koch, and Mira Mezini. Online reprogrammable multi tenant switches. In *Proceedings of the 1st ACM CoNEXT Workshop on Emerging in-Network Computing Paradigms*, pages 1–8, 2019.

[27] László Molnár, Gergely Pongrácz, Gábor Enyedi, Zoltán Lajos Kis, Levente Csikor, Ferenc Juhász, Attila Kőrösi, and Gábor Rétvári. Dataplane specialization for high-performance openflow software switching. In *Proceedings of the ACM SIGCOMM Conference*, 2016.

[28] Cheng Zhang, Jun Bi, Yu Zhou, Abdul Basit Dogar, and Jianping Wu. HyperV: A High Performance Hypervisor for Virtualization of the Programmable Data Plane. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 2017.

[29] Yu Zhou, Dai Zhang, Kai Gao, Chen Sun, Jiamin Cao, Yangyang Wang, Mingwei Xu, and Jianping Wu. Newton: Intent-Driven Network Traffic Monitoring. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2020.

[30] In-situ Programmable Behavioral Model. https://github.com/jijinfanhua/IPSA-ipbm.

[31] Yong Feng, Haoyu Song, Jiahao Li, Zhikang Chen, Wenquan Xu, and Bin Liu. In-situ programmable switching using rp4: Towards runtime data plane programmability. In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*, pages 69–76, 2021.

[32] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, et al. dRMT: Disaggregated Programmable Switching. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 1–14, 2017.

[33] Lavanya Jose, Lisa Yan, George Varghese, and Nick McKeown. Compiling packet programs to reconfigurable switches. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 103–115, 2015.

[34] J. Löfberg. YALMIP : A toolbox for modeling and optimization in MATLAB. In *In Proceedings of the CACSD Conference*, Taipei, Taiwan, 2004.

[35] Xilinx. Alveo U280 Data Center Accelerator Card. https://www.xilinx.com/products/boards-and-kits/alveo/u280.html.

[36] switch.p4. https://github.com/p4lang/switch/tree/master/p4src.

[37] Anirudh Sivaraman, Changhoon Kim, Ramkumar Krishnamoorthy, Advait Dixit, and Mihai Budiu. Dc. p4: Programming the forwarding plane of a data-center switch. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, pages 1–8, 2015.

[38] Hyojoon Kim and Arpit Gupta. ONTAS: Flexible and Scalable Online Network Traffic Anonymization System. In *Proceedings of the 2019 Workshop on Network Meets AI & ML*, pages 15–21, 2019.

[39] P4SRv6. https://github.com/ebiken/p4srv6.

[40] Jaeyoung Kim and Byungjun Ahn. Next-hop Selection Algorithm over ECMP. In *2006 Asia-Pacific Conference on Communications*, pages 1–5. IEEE, 2006.

[41] Sumet Prabhavat, Hiroki Nishiyama, Nirwan Ansari, and Nei Kato. On load distribution over multipath networks. *IEEE Communications Surveys & Tutorials*, 14(3):662–680, 2011.

[42] Cisco. Segment Routing over IPv6 dataplane. https://www.segment-routing.net/tutorials/2017-12-05-srv6-introduction/.

[43] Clarence Filsfils, Stefano Previdi, Les Ginsberg, Bruno Decraene, Stephane Litkowski, and Rob Shakir. Segment Routing Architecture. RFC 8402, July 2018.

[44] Charles Clos. A study of non-blocking switching networks. *Bell System Technical Journal*, 32(2):406–424, 1953.

[45] Václav E Beneš. On rearrangeable three-stage connecting networks. *The Bell System Technical Journal*, 41(5):1481–1492, 1962.

[46] Madihally J Narasimha. The batcher-banyan self-routing network: universality and simplification. *IEEE Transactions on Communications*, 36(10):1175–1178, 1988.

[47] LR Gokr and GJ Lipovski. Banyan networks for partitioning multiprocessing systems. In *Proc. First Annual Computer Architecture Conference*, pages 21–28, 1973.

[48] Kenneth E Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314, 1968.

[49] Spirent. Spirent spt-n4u compact chassis. https://www.spirent.com/assets/spirent_n4u_chassis_datasheet.

[50] Edgecore. Wedge 100bf-32x. https://www.edge-core.com/productsInfo.php?cls=1&cls2=5&cls3=181&id=335.

[51] Xilinx Vivado Design Suite. https://www.xilinx.com/products/design-tools/vivado.html.

[52] Behavioral Model of PISA (bmv2). https://github.com/p4lang/behavioral-model.

[53] Kun Cao, Junlong Zhou, Tongquan Wei, Mingsong Chen, Shiyan Hu, and Keqin Li. A survey of optimization techniques for thermal-aware 3d processors. *Journal of Systems Architecture*, 97, 2019.

[54] Wen-Wei Shen and Kuan-Neng Chen. Three-dimensional integrated circuit (3d ic) key technology: through-silicon via (tsv). *Nanoscale research letters*, 12(1):1–9, 2017.

[55] Xilinx. 3D ICs. https://www.xilinx.com/products/silicon-devices/3dic.html.

[56] Haoyu Song. Protocol-oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane. In *Proceedings of the second ACM SIGCOMM workshop on HotSDN*, 2013.

[57] Vector Packet Processing Platform. https://fd.io/vppproject/vpptech.

[58] M. Baldi. daPIPE: a Data Plane Incremental Programming Environment. In *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 1–6, 2019.

[59] Liangcheng Yu, John Sonchack, and Vincent Liu. Mantis: Reactive Programmable Switches. In *ACM SIGCOMM*, 2020.

[60] Teemu Koponen, Keith Amidon, Peter Balland, Martín Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Paul Ingram, Ethan Jackson, et al. Network virtualization in multi-tenant datacenters. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 203–216, 2014.

[61] J.D. Hadley and B.L. Hutchings. Design Methodologies for Partially Reconfigured Systems. In *Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*, 1995.

[62] Jiaxin Lin, Kiran Patel, Brent E. Stephens, Anirudh Sivaraman, and Aditya Akella. PANIC: A High-Performance programmable NIC for multi-tenant networks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.

A   rP4 Grammar

The rP4 grammar in Extended Backus-Naur Form (EBNF) is shown in Fig. 15, in which the non-terminals mutual to P4 are omitted.

<rp4program>	::=	*<header_defs>* *<strcut_def>* *<header_vector_def>* *<action_def>* *<table_def>* *<ingress_pipe>* *<egress_pipe>* *<user_funcs>*
<header_defs>	::=	'headers' '{' {*<header_def>*} '}'
<header_def>	::=	'header' *<headerName>* '{' *<structFieldList>* *<parser_def>* '}'
<parser_def>	::=	'implicit' 'parser' '(' *<headerFieldNameList>* ')' '{' {*<header_tag>* ':' *<header_name>*} '}'
<struct_defs>	::=	'structs' '{' {*<struct_def>*} '}'
<struct_def>	::=	'struct' *<struct_name>* '{' {*<member_type>* *<member_name>* ';'} '}' [*<alias_name>* ';']
<ingress_pipe>	::=	'control' 'rP4_Ingress' '{' {*<stage_def>*} '}'
<egress_pipe>	::=	'control' 'rP4_Egress' '{' {*<stage_def>*} '}'
<stage_def>	::=	'stage' *<stage_name>* '{' *<parser_mod>* *<matcher_mod>* *<executor_mod>* '}'
<parser_mod>	::=	'parser' '{' {*<instance_name>* ';'} '}'
<matcher_mod>	::=	'matcher' '{' {*<table_name>* ';'} '}'
<executor_mod>	::=	'executor' '{' {*<switch_tag>* ':' *<switch_actions>* ';'} '}'
<user_funcs>	::=	'user_funcs' '{' {*<func_def>*} 'ingress_entry' ':' *<stage_name>* ';' 'egress_entry' ':' *<stage_name>* '}'
<func_def>	::=	'func' *<func_name>* '{' {*<stage_name>*} '}'

Figure 15: rP4 EBNF.

Runtime Programmable Switches

Jiarong Xing   Kuo-Feng Hsu   Matty Kadosh[†]   Alan Lo[†]
Yonatan Piasetzky[†]   Arvind Krishnamurthy[‡]   Ang Chen

Rice University   [†]*Nvidia*   [‡]*University of Washington*

Abstract

Programming the network to add, remove, and modify functions has been a longstanding goal in our community. Unfortunately, in today's programmable networks, the velocity of change is restricted by a practical yet fundamental barrier: reprogramming network devices is an intrusive change, requiring management operations such as draining and rerouting traffic from the target node, re-imaging the data plane, and redirecting traffic back to its original route. This project investigates design techniques to make future networks *runtime programmable*. FlexCore enables partial reconfiguration of switch data planes at runtime with minimum resource overheads, without service disruption, while processing packets with consistency guarantees. It involves design considerations in switch architectures, partial reconfiguration primitives, reconfiguration algorithms, as well as consistency guarantees. Our evaluation results demonstrate the feasibility and benefits of runtime programmable switches.

1 Introduction

Programming the network to add, remove, and modify functions has been a longstanding goal in the networking community. Programmable switches [3, 8] represent the latest step toward this vision. Using high-level languages like P4 [3], network operators can customize packet processing behaviors at the switch program level. To change network processing, operators can deploy a different P4 program to the data plane, without the need for hardware changes or device upgrades. Programmable switches have enabled a host of new network applications in telemetry [15, 27, 35], measurement [40], security [39], and application offloading [18, 19].

Unfortunately, in today's programmable networks, the velocity of change is restricted by a practical yet fundamental barrier: switch functions are only programmable at compile-time, but they effectively become fixed functions at runtime. The switch program cannot be easily modified at runtime without reflashing the data plane hardware and carefully managing network-wide changes. To reprogram a network switch, operators need to first drain and reroute traffic from the target, install the new program image, and then redirect traffic back to its route. The error-prone nature of network maintenance procedures, the amount of manual coordination required, and the need to satisfy stringent SLAs pose severe constraints

on runtime program changes. To the extent that functions can be "hard-coded" in the device, they can be invoked for runtime response [41]. However, new functions that haven't been accounted for, or functions that cannot fit into the switch resources, are difficult to deploy at runtime. This stands in stark contrast to software data planes on host servers, where changes are easily accommodated and functions go through frequent upgrades [12]. The *ultimate* vision of programmable networks that seamlessly incorporates function changes *at any time* (e.g., based on traffic workloads or multi-tenancy requirements) still remains an elusive goal.

In this project, we pave the way toward *runtime programmable switches* by investigating the necessary building blocks and proposing concrete designs for each of them. FlexCore enables switch functions to be *continuously* programmable throughout the lifetime of the network. It develops a new set of control plane API to modify P4 program elements—match/action tables, control flow branches, and parsing graphs—while the switch data plane serves live traffic. These operations precisely instrument the switch program using *partial reconfiguration* primitives without affecting the rest of the data plane. This new modality of network programmability introduces an array of applications:

- Just-in-time network optimizations: When an optimization (e.g., network-accelerated multicast) is needed, it can be added just-in-time to serve the traffic workloads, and removed soon afterwards to keep the network lean.

- Real-time attack mitigation: If network attacks (e.g., DDoS, data exfiltration) are detected, we can inject mitigation modules exactly where needed; new attack patterns would trigger removal of expired modules and the insertion of new program components.

- Scenario-specific network extensions: A tenant can inject switch program extensions to the network. VM migration will carve out and graft the relevant program components to a different location of the network.

Also, telemetry applications do not have to commit to a fixed set of queries [27]; new network protocols can be added and removed dynamically; load-aware routing algorithms can be injected when needed [17]; different versions of switch programs can be deployed for canarying [42]. In fact, many (if not all) of today's programmable network applications will have more powerful, runtime programmable equivalents.

Achieving this goal requires a range of research challenges to be addressed: switch architecture designs that make runtime programmability natural, partial reconfiguration primitives for modifying live switch programs, atomicity and consistency guarantees on runtime changes, and algorithms for effectively computing reconfiguration plans. FlexCore makes contributions in all these dimensions.

Switch architecture. We base our FlexCore design upon a variant of disaggregated RMT (dRMT) [11]. dRMT separates switch memory from compute, and our architecture introduces another twist in its *partial* disaggregation design, where a small compute-local memory holds a indirection data structure that we call a *program description table* (PDT). This table contains metadata about the program control flow and is our target for reconfiguration. Decoupling program logic from its physical realization separates concerns: physical resources can be allocated and deallocated in scratch areas before program elements are modified for the changes to be visible.

Partial reconfiguration primitives. We develop a set of new primitives for adding, removing, and modifying program elements—this includes match/action tables, control flow branches and parser states. Unlike today's control plane API, which manipulates switch memory (e.g., adding/removing entries), the new API reconfigures switch compute.

Consistency guarantees. We propose three consistency guarantees for runtime reconfiguration: program consistency, element consistency, and execution consistency, with increasingly relaxed guarantees. These guarantees constrain the kind of "intermediate programs" that packets are allowed to encounter during partial reconfiguration. Program consistency states that all program modifications must take effect simultaneously. Element consistency is weaker, and states that modifications can be made visible in an element-by-element basis (e.g., one table at a reconfiguration step). Execution consistency is the weakest, but it still guarantees useful properties: packets never traverse execution paths that mix old and new program elements. In all cases, reconfigurations are atomic and do not disrupt data plane forwarding.

Algorithms. We develop algorithms for computing reconfiguration plans for different levels of guarantees.

Evaluation. We implement our design on a 12.8 Tbps merchant silicon (Nvidia Spectrum-2 SN3000 series), as well as a software simulator based upon bmv2. We evaluate the scalability of the reconfiguration algorithms and demonstrate a set of use cases in hardware and software platforms, showing that FlexCore enables a truly adaptive network core.

2 A Case for Runtime Programmability

The quest for network programmability has been an important undertaking in the community. Network switches used to be blackboxes, with opacity at both control and data planes. OpenFlow SDN opened up the control plane for programmatical control, and as of late, programmable data planes enable flexible packet processing pipelines without hardware upgrade. Operators can customize the data plane by removing unnecessary switch functions or adding new ones at the program level. P4 switch programs are compiled into a binary image, which is flashed to data plane hardware for deployment. Researchers have seized this opportunity to systematically rearchitect network telemetry [15, 27, 35], measurement [40], security [39], and application offloading [19].

However, today's programmable data planes have a notable limitation—they cannot be reprogrammed at runtime. If an operator can anticipate all required functions at compile time, and if these functions can fit into the switch resource constraints, then they can be combined and deployed together in the switch. But once deployed, the switch is committed to the hardcoded behaviors as specified at compile time, until the next program reflash. At runtime, only 'micro' changes are permitted, such as modifying flow table entries or register values from the control plane. This affords some flexibility [41]; however, as macro-level program logic changes are hard to make, accommodating requirements that truly arise 'on-demand' (e.g., security incidents) remain an elusive goal. Also, since switches have constrained resources, even if we had an 'oracle' planner that anticipates all needed functions, they may not fit into the switch together at compile time.

To remedy this problem, we need runtime programmable switches. This not only enables new use cases as motivated above, but also calls for a rethink as to how networks can be specialized. The operator can, at any point in time, aggressively optimize the network data plane to only retain a minimal amount of processing logic. This reduces switch resource footprints, improves network energy efficiency, and also keeps network latency at a minimum. If extra functionality is required, the program elements can be injected precisely where and when they are needed. If a functionality is no longer in use, it can be removed to ensure that the data plane stays at its leanest. Viewed from the lens of the classic 'end-to-end' arguments [31], in-network processing no longer incurs a common overhead to all applications.

3 The FlexCore Switch Architecture

Our switch architecture adopts a disaggregated RMT model [11], where compute resources (i.e., match/action processors) are split from memory (i.e., SRAM/TCAM), and they are interconnected via a crossbar. Each MA processor holds a copy of the P4 program, and processes packets in a run-to-completion manner.

In the RMT architecture [8], each stage contains a slice of compute and memory resources that cannot be reassigned to other stages. This tight coupling makes runtime reconfigurations challenging. For instance, inserting an MA table to a stage may require device-wide table shuffling and reallocation to make space. Removing an MA table from a stage will leave 'holes'—fragmented resources that cannot be easily reused by other program elements. These operations can be intrusive.

A disaggregated architecture, on the other hand, breaks
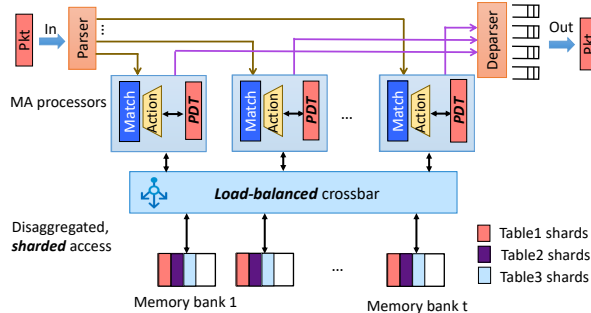
Figure 1: The FlexCore switch architecture. Highlighted in bold+italic are the customizations to dRMT [11].

resource allocation boundaries and enables reconfigurations to be performed locally—i.e., it enables *partial reconfiguration*. If a reconfiguration releases a table, the deallocated resources can be dedicated to *any* other program elements irrespective of their 'locations'. New tables can be inserted to any part of the program without having to change existing resource allocation decisions. Similar properties hold for resources that implement control flow branches and parsing graphs.

dRMT customizations. Our silicon also implements several customizations for performance, flexibility, and usability. Figure 1 shows the high-level architecture.

(i) Sharded resource allocation. In the dRMT architecture, an MA table is allocated in one specific SRAM/TCAM bank. Simultaneous accesses to the same table (or different tables in the same memory bank) from different processors creates contention at the crossbar. In FlexCore, all tables are *t*-way sharded, where *t* is the number of memory banks. When inserting a table entry, FlexCore first computes a hash *h* from the match key as the shard ID, and then allocates the entry in the *h*-th SRAM/TCAM bank. When performing a match, the same hash function is computed to retrieve the shard ID. This allows FlexCore to sustain linerate without complex mechanisms to detect and avoid access contention. The crossbar is always load-balanced and has uniform access patterns.

(ii) Hybrid programmability. Our switch exports a set of fixed-function ASIC modules as common building blocks (e.g., L2 bridging, L3 routing). These functions can be called by or bypassed from the P4-programmable logic. The fixed blocks are more resource- and energy-efficient, as their implementations are heavily-optimized, hardwired ASIC. By providing these building blocks, P4 programmers don't need to redevelop them from scratch. Moreover, they also represent a minimum "baseline" program that, if necessary, traffic can always fallback on during reconfiguration.

(iii) Indirection. FlexCore employs a partially disaggregated design, where each processor has a small amount of local SRAM to store a special *program description table (PDT)* for indirection. Accesses to PDT do not go through the crossbar and enjoy lower latency. PDT stores the 'program skeleton'—the control flow graph—and decouples the control flow operations from main SRAM accesses. Our partial recon-
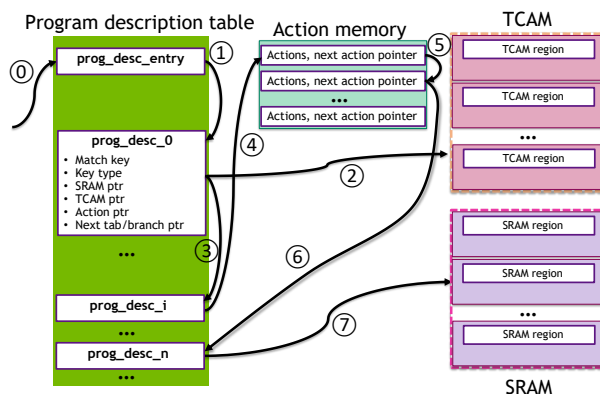
Figure 2: Runtime reconfigurable tables and control flow branches, the indirection mechanism via the program description table (PDT), and an example execution as illustration.

figuration mechanisms make heavy use of the PDT to modify program elements. Similarly, a parser state table (PST) serves as indirection for the parsing hardware.

4 Runtime Reconfiguration Primitives

FlexCore introduces a set of novel primitives that, when invoked by the control plane, partially reconfigure a P4 switch program. These primitives operate on a graph representation of a P4 program by adding, removing, or modifying nodes and edges. In a P4 program, the match/action logic is captured by the 'table flow graph' [8], where nodes represent MA tables or conditional branches (realized in table-independent actions), and edges represent non-conditional, table dependency control flow. For the parser logic, the nodes represent parser states (which also contain header extraction rules), and the transition rules are the edges. Next, we first describe the primitives on the table flow graph and then the parsing graph.

4.1 Program description table

A key indirection data structure that enables partial reconfiguration of the table flow graph is what we call a *program description table (PDT)*, as shown in Figure 2. Each match/action processor maintains a local PDT and it is dedicated to a specific switch port. All packets arriving at a port will first hit a default entry in the PDT to activate packet processing.

The entries in the PDT are compiled from a P4 program. Each entry stores metadata about a *program element*, which could be a match/action table or a table-independent ALU action that implements conditional control flow. The metadata contains entry type, match key/type, and a resource pointer that refers to the physical realization of that program element. The pointer address could be an SRAM location (for exact and algorithmic ternary matches), a TCAM location (for ternary matches), or an action location (for conditional branches)—with a 'union' semantics as only one pointer type can be valid for a PDT entry. The address is specified by the base address of a memory region, the size of the region, as well as the offset from the base address. Each PDT entry also contains a 'next'

pointer, which encodes unconditional control flow to the next program element (i.e., MA table or conditional branch).

This indirection provides several advantages for runtime programmability: a) operations for adding and removing a program element are decoupled from resource allocation operations, as the first occur in the PDT and the second in the memory regions; b) PDT entries serve as a local scratch—entry modifications are lightweight and do not touch switch-wide shared resources, and they can be changed in a transactional manner. The PDT enables runtime reconfiguration of match/action tables and the control flow graph, which we discuss next.

4.2 Runtime reconfigurable tables

MA tables are the key processing elements in a P4 program. FlexCore enables the addition and deletion of tables using several partial reconfiguration primitives.

Allocation+deallocation. ALLOCTBL(T) allocates a new table, and DEALLOCTBL(T) deallocates an existing one. Both are control plane operations that have a centralized view of PDT tables, and they accept the table definition T as the input argument. Allocations first identify free slots to create new PDT entries. In a new PDT entry, the match key and type are filled in with the specified table attributes. SRAM and TCAM resources are then allocated based on the table attributes, and both are sharded across all memory banks. Finally, the control plane fills in the resource pointer, finishing the table allocation. Deallocations could directly remove the entry and its resources, or it may defer their removal to a later garbage collection phase. (Actual table entries are added/removed just like in today's switches, via existing control plane API such as that defined in P4Runtime [5].) Importantly, allocation/deallocation operations are not visible to network traffic until we invoke insertion/deletion primitives.

Insertion+deletion. Changes are made visible via another primitive: SETPTR(T,NXT) modifies T's next pointer to NXT. Table insertions invoke multiple SETPTR calls to place T in the program; deletions perform the opposite operations. Insertions must happen after resources have been allocated, and deletions before deallocation. Each pointer change is atomic in hardware. (To ensure atomicity for a collection of changes, we need another mechanism called a 'flex branch' as discussed later.) Insertions and deletions alter the view of the program state from the perspective of network traffic.

4.3 Runtime reconfigurable control flow

Conditional branches are implemented in ALUs as table-independent actions. Like tables, a conditional branch takes up one PDT entry, but its resource pointer addresses the action memory instead of SRAM/TCAM. In addition, the PDT entry for a conditional branch has a null 'next' pointer; its two jump addresses are instead encoded in the ALU action, one for each branch condition. N-way conditionals are implemented as cascading binary branches. Control flow branch
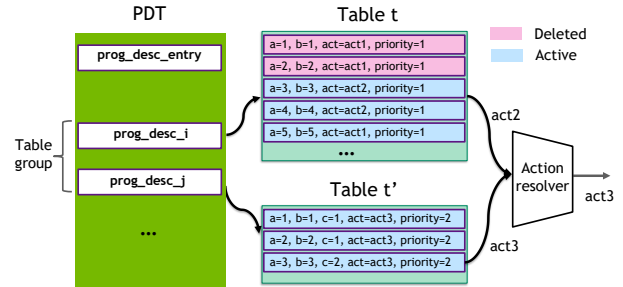
Figure 3: Primitives for in-place table modification.

modifications are performed using the following primitives.

Allocation+deallocation. FlexCore introduces a primitive, ALLOCCOND(B, PRED, BR$_1$, BR$_2$), to allocate a control flow branch based on PRED, where BR$_1$ and BR$_2$ are the jump addresses for the true and false branches, respectively. Allocation of an N-way conditional is performed by successive invocations of ALLOCCOND with cascading jump addresses. A predicate PRED corresponds to an ALU action that checks the condition and produces a true/false evaluation. This binary result is consumed by a hardware 'goto' microcode that jumps to the next program element. If PRED evaluates to true, 'goto BR$_1$' directs the control flow to the next table or a cascading branch; otherwise, it branches to BR$_2$. Deallocations free action memory and PDT entries.

Insertion+deletion. A conditional branch can be activated by a) SETPTR(T,B), which points a table's next pointer to the new branch B, and b) SETCONDPTR(B,N$_1$,N$_2$), which sets one or both of the jump addresses of a branch. In the case where SETCONDPTR modifies two pointers, the operation is not atomic. Atomicity is achieved similarly using 'flex branches' that we will discuss later. Deletions achieve opposite effects.

4.4 In-place table modifications

So far, all primitives that we have described can be used at any level of consistency guarantees. In this and the next subsections, we describe two special sets of primitives for table modifications and parser reconfigurations as well as their respective consistency properties.

Table modifications can be performed by adding a new table and deleting the old, in which case the intermediate state has size $2 \times |T|$ (assuming both tables have size $|T|$). But FlexCore also exposes a more efficient primitive to reformat a table in-situ with an intermediate state of $|T|$. A MODTBL(T,T') primitive reformats T using the definition as specified in the new table definition T', which could include new match key/type and actions. This is achieved by a PDT mechanism called *table groups*. Several PDT entries can be 'grouped' together and processed in parallel at the MA processor. MODTBL creates a new PDT entry using T' and groups it with the entry for T. It then gradually moves entries from T to T', reformatting each entry using the new key or action, and setting the entries in T' with higher priority. In this transient state, the MA processor looks up both tables and resolves
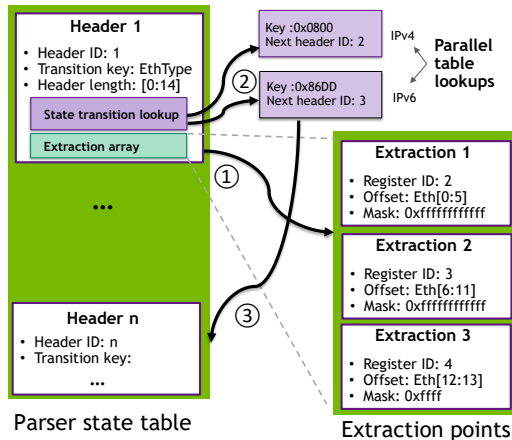
Figure 4: Runtime reconfigurable parsers and the indirection mechanism via the parser state table (PST).

them using an *action resolver* that chooses the higher-priority result. When T become empty, the PDT entries are de-grouped and T gets deleted. MODTBL triggers simultaneous applications of parallel tables, so this mechanism is different from the 'flex branch'. We will discuss its consistency guarantees later in Section 4.6.

4.5 Runtime reconfigurable parsers

Header parsing logic requires different mechanisms for reconfiguration. We describe the parser hardware next, and then the reconfiguration primitives for the parser graph.

The parser state table (PST). Figure 4 presents the hardware architecture for the reconfigurable parser. The key indirection data structure is a *parser state table (PST)*, which stores an array of parser states. Each entry stores a) parsing information for that header, b) an extraction array that extracts header fields, as well as c) a parallel transition lookup component that determines the next state based on the current header values. Similar as the PDT, this indirection ensures that state additions and removals are easily achieved at runtime.

The PST implements a finite state machine, where each entry represents one state and contains transition rules to other states. This array is indexed by a logically assigned header ID that starts with one and ends with the maximum state ID as constructed from the program. When a packet comes in, it first matches against the default entry (ID=1) for parsing. At every step, the hardware uses the 'header length' and 'transition key' defined in the current entry (as well as a base register that remembers how much data has been parsed) to identify the correct offset into the packet. A chunk of data of the size 'header length' is then sent to extraction logic, which uses shift-and-mask to further segment the data chunk into multiple fields (e.g., EtherType, SMAC, DMAC) of varied sizes. These extracted fields are stored in an *extraction array* that is associated with the current header entry. These are further combined using a recombiner into a PHV (packet header vector) and streamed to the ingress blocks.

Simultaneously with header extraction, FlexCore uses a parallel set of logic to identify relevant headers to compute the next parsing state. This relies on a similar extraction logic but does not materialize header fields in the extraction array. Rather, it uses the preconfigured 'transition key' to perform a parallelized lookup. It muxes the key through a lookup table that contains all transition rules as compiled from the parser—e.g., IPv4 packets transition to ID=2, and IPv6 to ID=3. A demux combines the lookup results from all rules and computes the next state ID. Parsing continues until it encounters an accept state, at which point the extracted headers are sent to the ingress logic for MA processing.

Reconfiguration primitives. Runtime parser reconfiguration modifies the parser states, extraction rules, and transitions. FlexCore exports ALLOCSTATE(S), ALLOCTRANS(S1,S2), and ALLOCEX(R) for allocation of new states, transitions, and extraction rules, respectively. ALLOCSTATE(S) creates a new PST entry and the respective transition key and header length. ALLOCTRANS(S1, S2) sets up transition rules in the transition matching mux and demux. ALLOCEX(R) sets up an extraction rule in a parser state that locates a certain offset in the current header and outputs the result to an extraction register. Each primitive has its DEALLOC analogue.

Edits to the transition rules with ALLOCTRANS are immediately visible to network traffic, so for multiple changes, FlexCore requires the parser diff or the new parser to be prepared in PST scratch, before they are activated together in a single atomic step. Otherwise, network traffic will be parsed with a mix of old and new parsing logic. In the current hardware, parser changes are only possible with 'program consistency', which, as we will discuss later, requires higher resource headroom to maneuver. This limitation stems from the lack of a 'flex branch' equivalent in the current parser hardware, which is necessary for using the version metadata for transactions. In future hardware generations, this can be incorporated by adding transition version numbers as well as match logic using the versions.

4.6 Summary

We now discuss the two special-case primitives: table modifications and parser changes. MODTBL relies on 'table groups' instead of 'flex branches'. When a MODTBL operation is in progress, it guarantees that each packet is only processed with the old or new version of the table; in this sense, the intermediate states as seen by the packets satisfy 'execution consistency'. However, MODTBL cannot be parallelized with other program modifications, as 'table groups' do not atomically control which version is encountered by packets. Parser changes, on the other hand, satisfy program consistency; but the current hardware doesn't support weaker guarantees, which require 'flex branches'. In the next section, our reconfiguration algorithms primarily focus on changing MA tables and control flow branches, where all three consistency guarantees apply and are achievable at different overheads.
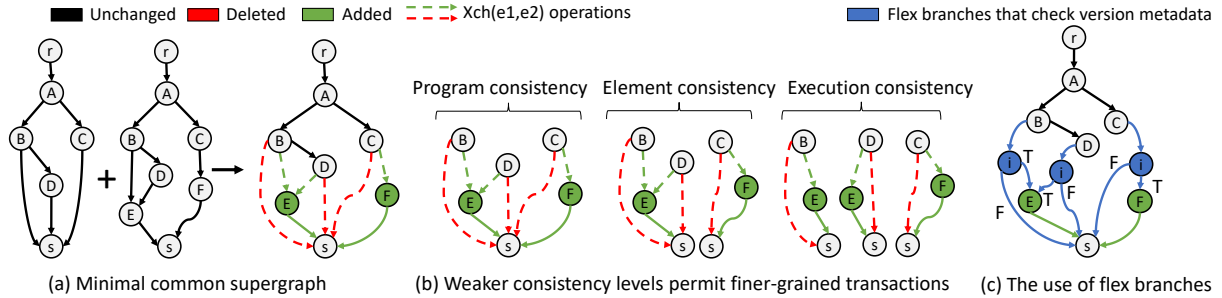
Figure 5: (a) FlexCore constructs a minimum common supergraph between two programs. (b) Weaker consistency guarantees reduce resource requirements for reconfigurations, and allow more intermediate states to be exposed to network traffic. (c) To ensure atomicity, FlexCore inserts 'flex branches' that can branch to the old or new versions depending on the version metadata. These branches are deleted after reconfiguration completes. Nodes A-F represent MA tables or conditional control flow branches. Virtual nodes r and s are added as the sources and sinks of the DAGs, respectively. Virtual nodes i denote flex branches.

5   Runtime Reconfiguration Algorithms

The FlexCore reconfiguration algorithms rely on the partial reconfiguration primitives to transform an existing switch program *prog* to a new one *prog**. We represent each P4 program as a directed acyclic graph (DAG), G for *prog* and G^* for *prog**. Nodes are the MA tables and conditional branches, and edges represent unconditional dependencies (or packet dataflow through the program). Our goal is to compute a *reconfiguration script* [9], a series of graph edit operations to nodes and edges to transform G into G^*. We denote the reconfiguration sequence as $G \to S_1 \to \cdots \to S_n \to G^*$, where $S_i, in \in [1..n]$ are the intermediate DAGs and each step from S_i to the next state is atomic. Depending on whether (or what types of) intermediate states are allowed to be exposed to network traffic, we propose three levels of consistency guarantees: program consistency, element consistency, and execution consistency, with a decreasing order of strictness. Stronger guarantees are achieved by preparing larger portions of the program diff in scratch memory, requiring that the switch resources must have enough slack for the reconfiguration. Weaker guarantees allow FlexCore to operate within more restricted headroom. Figure 5 includes an illustrative example.

5.1   Program consistency

This is the strongest level of consistency guarantees: no intermediate state is exposed to any packets. The switch program as encountered by network traffic is either G or G^*. This is important for any scenario that requires strong network processing guarantees, where exposing intermediate state would cause operational disruption. For instance, a load balancer or NAT may contain two match/action tables, one for mapping DIP to VIP and another for the reverse direction [25]. Updates to the program (e.g., rehashing) should not take effect until both tables have been reconfigured.

Program consistency. *A sequence $G \to S_1 \to \cdots \to S_n \to G^*$ achieves program consistency if the following property holds for all $S_i, i \in [1..n]$. For any element t (node or edge) in S_i, if $t \in G^*$ and $t \notin G$, then $S_i = G^*$. Similarly, for any element t*

in S_i, if $t \in G$ and $t \notin G^*$, then $S_i = G$.

Put in simpler terms, reconfigured program elements aren't visible to network traffic until all reconfigurations finish: an "all-or-nothing" guarantee. To achieve this, all edits must be prepared in an 'offline' scratch area. They are made visible in an atomic transaction that, from the packets' perspective, changes G to G^* in one single step. Without the partial reconfiguration primitives in FlexCore, one would need to instantiate the entire program *prog** in the scratch while the old one *prog* is still active. Therefore, the switch resources must have enough slack to accommodate the co-existence of both programs—i.e., there must be a headroom of $|G|+|G^*|$. Supposing that $|G| \approx |G^*|$, then the switch resource utilization must be kept to $\leq 50\%$ for runtime changes to be feasible. This is a stringent requirement.

Algorithm. Our new primitives enable FlexCore to only prepare the 'diff' while reusing shared program elements, so the switch only needs to accommodate $|G|$ and newly inserted elements of the size $\Delta \ll |G^*|$. In order to compute the diff, FlexCore merges two DAGs G and G^* into a *minimum common supergraph (MCS)* [9]. An MCS is the union of the input DAGs that minimizes the diff as caused by mismatched elements. In our context, only nodes take up resources and edges are pointer fields in the nodes and do not consume physical resources; so our MCS algorithm primarily extracts node-level diff. Using this MCS, we compute a set of edit operations as our reconfiguration script. INS(v) and DEL(v) inserts and deletes a node, respectively; and INS(e) and DEL(e) operate on edges. A special edge substitution operation XCH(e,e′) is allowed if both edges share the same source node and are of the same type (i.e., both are 'next' pointers or both are true/false jump addresses). In terms of resource overheads, INS(v) reduces and DEL(v) increases switch headroom by $|v|$, respectively, where $|v|$ is the table size (for MA tables) or action memory size (for conditional branches). Edge operations do not affect resource headroom. Figure 6 shows the algorithm, which colors the MCS: shared elements in black, new elements in green, and deleted elements in red.

```
function PROGRAMCONSISTENCY(prog, prog*)
    // Compute minimum common supergraph
    G ← GETP4DAG(prog); G* ← GETP4DAG(prog*)
    G ← MERGEDAGS(G, G*)
    // Compute reconfiguration script
    Script ← ∅
    for node or edge t ∈ G do
        if t ∈ G ∧ t ∈ G* then
            COLORBLK(t)
        else if t ∈ G ∧ t ∉ G* then
            COLORRED(t); Script.Add(DEL(t))
        else if t ∉ G ∧ t ∈ G* then
            COLORGRN(t); Script.Add(INS(t))
    Script.IdentifyEdgeXch(G)
    return Script
```

Figure 6: The program consistency algorithm.

Atomicity. To ensure that intermediate states are not visible until all reconfigurations complete, FlexCore groups the edits in a transaction to achieve atomicity. We use a hardware mechanism that we call a *flex branch*. During the transaction, inserted program elements are guarded by an extra conditional branch that implements a check on special version metadata: 'if (meta.v==0)' branches to the old program elements and 'if (meta.v==1)' to the new. Deletions are also guarded by flex branches instead of being deleted right away. The transaction is committed when FlexCore modifies the version metadata, after which deleted elements can be safely removed.

5.2 Element consistency

A relaxed consistency guarantee, which allows reconfigurations to proceed within more restricted headroom. In program consistency, preparing the diff in scratch area leads to a resource spike of Δ. Therefore, in order to accommodate runtime reconfigurations, the switch utilization must be upper-bounded to leave sufficient headroom Δ.

Element consistency breaks the reconfiguration into several finer-grained transactions that can be performed with lower headroom δ ≪ Δ. This allows FlexCore to drive up switch utilization even further while still preserving the ability to make runtime reconfigurations. Every smaller transaction will add and remove certain program elements, with the goal of releasing some switch resources to accommodate subsequent transactions. Under this guarantee, intermediate states can be exposed to traffic, but only if there is a consistent view as to which program elements have been updated (inserted or deleted). If program elements (nodes or edges) are reachable from each other, they must be updated together. Unreachable edits are partitioned to different transactions as they are independent from the view of network traffic. This property is useful when program updates can be applied incrementally with well-defined semantics. For instance, a firewall that uses independent ACL tables for different types of traffic (e.g., TCP vs. UDP) can be added or removed on a table-by-table basis. A traffic normalizer [1, 22, 39] may apply different

```
function ELEMENTCONSISTENCY(prog, prog*)
    // Compute overall script
    Script ← PROGRAMCONSISTENCY(prog, prog*)
    // Reachability analysis. Optimization using Xch operations.
    for all Xch(u→v, u→v′) ∈ Script do
        u.Reachability ← DFS(u, G)
    // Partition script by reachability
    Partitions ← INITPARTITIONFOREACHEDIT(Script)
    while ∃ reachable partitions p, q do
        MERGEPARTITIONS(p, q)
    return Partitions
```

Figure 7: The element consistency algorithm.

security functions for incoming and outgoing traffic—e.g., normalizing TTL fields for incoming packets, but clearing TCP options for outgoing ones.

Element consistency. *For any intermediate state $S_i, i \in [1..n]$, we require the following properties to hold. For any element t in S_i, if $t \in G^*$ and $t \notin G$, then for any other element t' in G^* where $t' \rightsquigarrow^* t$ (i.e., t' can reach t in G^*) or $t \rightsquigarrow^* t'$, we require that $t' \in S_i$. Similarly, for any element t in S_i, if $t \in G$ and $t \notin G^*$, then for any other element t' in G where $t' \rightsquigarrow t$ (t' can reach t in G) or $t \rightsquigarrow t'$, we require that $t' \in S_i$.*

Stated simply, if a new program element is visible in the intermediate state, it should be visible to all packets that traverse this element in the new program, even if they follow different execution paths through the program. A deleted element is no longer visible to packets regardless of their execution paths.

Algorithm. As Figure 7 shows, we first invoke the program consistency algorithm to compute the overall reconfiguration script, and then partition this script into independent, smaller transactions. This relies on a DFS search on **G** to compute whether one edit may affect another. If two edits operate on unreachable regions of the graph, they may proceed independently; otherwise they belong to the same partition. Initially, each edit is in its own partition. Partitions are merged if they are reachable from one another—*p* and *q* are said to be reachable if their edit operations involve elements that are reachable in either direction in **G**. This implies that the algorithm scales quadratically with the number of edit operations.

Although we can perform DFS from all nodes and edges in **G** in polynomial time, in practice we only need to do so from nodes that are involved in an XCH operation. This computes all needed reachability information to merge the partitions, because such nodes are the boundaries between the new and old graphs. Red nodes/edges are reachable from at least one such XCH node by following its red outgoing edges, and similar properties hold for the green color. When no further merges are possible, the algorithm returns a partition of the reconfiguration script.

Atomicity. Each smaller transaction begins with 'meta.v==0'. Flex branches guard intermediate changes or make them visible by changing 'meta.v'. The reconfiguration finishes after all constituent transactions are committed.

```
function EXECUTIONCONSISTENCY(prog, prog*)
    // Compute overall script
    Script ← PROGRAMCONSISTENCY(prog, prog*)
    // Bounded reachability analysis
    for each Xch(u→v, u→v') ∈ Script do
        Xch.Reachability ← BOUNDEDDFS(u, G)
        INITSUBPARTITION(Xch.Reachability)
    // Order partitions
    for each Xch1, Xch2 ∈ Script do
        if Xch1⤳*Xch2 then
            ADDCONSTRAINT(Xch1 ≥ Xch2)
        if Xch1⤳Xch2 then
            ADDCONSTRAINT(Xch1 ≤ Xch2)
        if Xch1 ≤ Xch2 ∧ Xch1 ≥ Xch2 then
            MERGESUBPARTITIONS(Xch1, Xch2)
    Subpartitions ← CONSTRAINEDSORT(Subpartitions)
    Subpartitions ← DEDUPEDITS(Subpartitions)
    return Subpartitions
```

Figure 8: The execution consistency algorithm.

5.3 Execution consistency

We next consider an even more relaxed guarantee with more finer-grained transactions. Under execution consistency, a new program element may only be visible to some execution paths but not others. Likewise, if an element is deleted from some execution paths, other executions may still use this element until all reconfigurations finish. Such intermediate states are still consistent in that a packet never experiences an execution path that mixes old and new elements. This is the weakest level of consistency that we consider in FlexCore. It is a suitable guarantee for program changes that are in nature non-disruptive—e.g., functions that do not interfere with packet processing decisions, or functions where inaccuracy is tolerable. For instance, a telemetry module that samples or aggregates traffic can be added or removed using execution consistency. The intermediate states merely introduce noise to the monitoring data, but do not break functionality.

Execution consistency. *For any intermediate state $S_i, i ∈ [1..n]$, any execution path through this program, $p ∈ S_i$, should satisfy that $p ∈ G$ or $p ∈ G^*$.*

This allows reconfigurations to proceed at a per execution path basis. Paths are added to the program as a whole, or they are deleted as a whole. But packets will not encounter partial paths or paths that mix old and new elements.

Algorithm. Figure 8 shows the pseudocode. As before, we perform a reachability analysis from Xch nodes; but unlike in element consistency, the DFS terminates when encountering other Xch nodes or shared (black) nodes. The visited elements form a subpartition for each Xch node. In element consistency, if Xch1 reaches Xch2, they are merged into the same transaction. But execution consistency only requires the merge of certain Xch regions, but not all. If independent reconfigurations of Xch1 and Xch2 do not lead to partial or mixed paths, then their edits can be performed separately.

Specifically, we analyze the ordering relation between all pairs of Xch nodes. If Xch1 can reach Xch2 via a green (new) path p_g, then reconfiguring Xch1 before Xch2 will lead to a situation where the part of p_g in Xch1 is activated but its extension into Xch2 is not, leading to a mixed path. Reconfiguring Xch2 before Xch1, on the other hand, is safe because the changes are not reachable from Xch1. Of course, this reconfiguration will not enable p_g, but this may enable other paths elsewhere so it is a valid plan to be considered. Similarly, if Xch1 reaches Xch2 via a red (old) path p_r, then reconfiguring Xch2 before Xch1 will delete p_r from its end while its earlier part is still in use, resulting in mixed colors. Reconfiguring Xch1 before Xch2, on the other hand, is valid because it simply removes p_r. If Xch1 can reach Xch2 via green and red paths, then the only valid plan is to reconfigure both regions atomically.

This above ordering relation generates a set of constraints across Xch nodes, as well as an ordered set of subpartitions. These subpartitions are finer-grained than the partitions in element consistency, so they enable smaller transactions. One final care must be taken: since subpartitions may be reachable from each other, the bounded DFS may reach shared elements from different Xch nodes. The edit operations in two subpartitions, therefore, may have overlaps. A deduplication step over the subpartitions ensures that a deletion operation is deferred to the last subpartition where the deleted element is used, and that an insertion operation is performed in the first subpartition where the new element occurs. This concludes the execution consistency algorithm, whose complexity is quadratic with regard to the number of Xch nodes.

Atomicity. The use of flex branches makes each subpartition visible to network traffic atomically. The entire transaction finishes when all subpartitions have been reconfigured.

5.4 Summary

The reconfiguration script is then realized by the partial reconfiguration primitives in Section 4—e.g., an operation on v will translate into a table or branch operation depending on v's type. For program consistency, all edits are applied in one single, atomic step, but for element and execution consistency, the (sub)partitions are applied sequentially. This raises another consideration as to the ordering of the transactions in the latter two algorithms to minimize the maximum utilization peak. We perform an exhaustive search over the order. This search terminates when it has identified a feasible order or when it concludes that no such order exists.

6 Limitations and Discussions

Program equivalence. The FlexCore partial reconfiguration primitives and algorithms operate on P4 program elements, relying on the structural differences between two P4 programs. It currently doesn't analyze whether structurally different programs may have the same semantics [10, 13], which is an interesting avenue for future work.

Stateful packet processing. FlexCore currently does not support stateful switch programs. The P4 standard defines persistent state as an "extern" feature that is up to the individual architectures to implement (e.g., registers in PSA). Partial reconfiguration of stateful features raises additional questions as to how network state should be ported to the new program, e.g., with programmer-supplied state transformation functions, much like in SDN software controller upgrades [32].

Resource headroom. The FlexCore algorithms require that the switches have sufficient resource slack, but there could be scenarios where even the weakest consistency would require more resource headroom than available. To address this, one could relax execution consistency even further to capture which types of 'mixed' executions are still semantically meaningful; alternatively, one could also migrate certain resources to other devices to make room for the reconfiguration.

Other architectures. The FlexCore primitives target P4 program changes, so they are in principle architecture-independent. The dRMT variant that FlexCore uses makes runtime reconfiguration particularly natural, but most P4 targets have some degree of runtime flexibility. The RMT architecture, for instance, may be augmented with the ability to reconfigure each stage independently. Software switch targets (e.g., for the host or NIC) expose even more runtime flexibility than switch ASICs. Although the original dRMT project [11] didn't provide an ASIC implementation, we believe that our indirection structures are compatible with its outlined design.

Other languages. FlexCore's reconfiguration primitives target P4 programs, but for other languages (e.g., NPL [2], PoF [36]), one should be able to develop analogous reconfiguration primitives based upon their respective language features. The property of runtime programmability is not tied to a specific language.

7 Implementation

We have implemented FlexCore in several components. The reconfiguration primitives are implemented by manipulating the hardware ASIC control registers via the PCIe interconnect from the control plane. The indirection structures are implemented in the Spectrum-2 silicon design, and FlexCore is the first effort to leverage them for runtime, partial reconfiguration. Our compiler uses p4c [4] as the frontend; it implements incremental compilation of P4 program elements, generating an individual binary image for each component, instead of outputting a monolithic binary for the entire program. The consistency algorithms are implemented at the control plane.

The hardware cost to enable runtime programmability comes from the use of indirection structures, including the PDT and PST. The PDT supports full reconfigurability at all consistency levels, but the PST only supports program consistency. We estimate the cost of the current PDT and the cost for making the PST fully reconfigurable at runtime.

Each MA processor has a local PDT, which holds roughly 1k entries—i.e., the largest P4 program it supports should have no more than ∼ 1k MA tables and conditional branches. The ASIC supports up to 128 MA processors overall. Recall the PDT format as shown in Figure 2: each entry contains a) a description of the match key, b) entry type (SRAM/T-CAM/Actions), c) resource pointer, and d) next table/branch pointer. In the worst-case scenario, each MA table has a different key, resulting in 1k distinct keys that the switch needs to support; this requires 10 bits to represent each distinct key in the PDT entry. The entry type field distinguishes between three types, requiring 2 bits. The resource pointer requires 20 bits, which is able to index one million distinct memory lines for SRAM/TCAM/Actions, roughly 20MB in size (the 'main database'). The next table/branch pointer requires 10 bits to index another PDT entry as the next hop. Overall, each PDT entry requires 42 bits, each PDT table consumes 5.25kB for 1k entries, and across 128 PDT tables the hardware overhead is 0.67MB, or 3.3% of the main database. The flex branch mechanism is implemented using existing ALUs, so it doesn't require dedicated hardware. For the PST, the Spectrum-2 parser hardware only supports runtime reconfiguration at program consistency level. This does not contain the 'flex branch' equivalent and the 'version' support for transition rules, which would be necessary for other consistency levels. We estimate the overhead of these additional structures to be under 1% of the main database.

8 Evaluation

We present a comprehensive evaluation of FlexCore, by applying our design to a 12.8 Tbps hardware ASIC and also a software simulator (a fork of bmv2) that has been integrated with the same reconfiguration primitives. To evaluate scalability, we have used a set of synthetic and real-world P4 programs. To synthesize the P4 corpus, our tool takes a specified program size and generates a random control flow graph. For real-world programs, we have used switch.p4, NetCache [19], and NetHCF [23], which represent large, medium, and small programs, respectively. The program edits are also generated randomly, which may mutate, insert, remove, or swap program elements. The edits are controlled by a parameter α, the reconfiguration ratio. If a program has 100 program elements, and a reconfiguration adds, removes, or exchanges 10 of them, we say that α = 10%. To evaluate realistic reconfiguration scenarios, we perform case studies using switch-based multicast, telemetry, attack mitigation, and tenant-specific extensions, on hardware and software platforms.

8.1 Reconfiguration primitives

We start by measuring the number of hardware operations that each reconfiguration primitive involves. These primitives are invoked by the control plane, and they modify a memory-mapped region of the PCIe device (i.e., the data plane). The PCIe bus sustains a peak throughput of ∼1 million operations per second. The control plane, however, is bottlenecked by the software speed; each operation took several milliseconds to
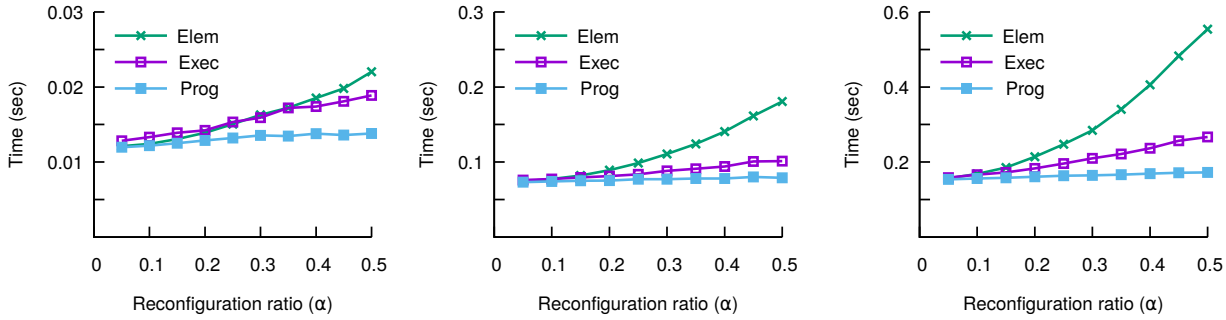
(a) NetHCF (V=43, E=58).　　(b) NetCache (V=109, E=129).　　(c) switch.p4 (V=168, E=242).

Figure 9: Scalability of FlexCore on three real-world programs. V: number of nodes, E: number of edges.
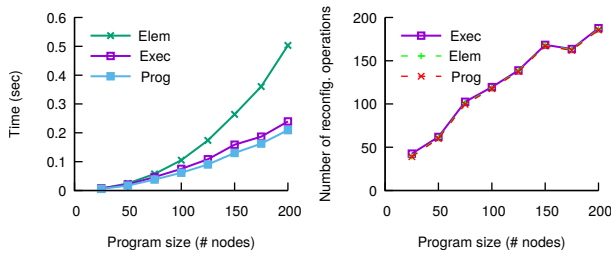
(a) Turnaround time　　(b) Number of primitives

Figure 10: The FlexCore algorithms scale well.

complete with software overhead. Table 1 shows the number of hardware register DWORD writes for each reconfiguration primitive. As we can see, table operations are the most heavyweight, control flow branch operations follow, then parser operations, and finally, edge edits complete within one write and are atomic. Deallocations have the same number of operations as their allocation analogues.

Primitive	RegAccess	Primitive	RegAccess
ALLOCTBL	112	GROUPTBL	112
ALLOCCOND	43	ALLOCSTATE	22
ALLOCTRANS	5	ALLOCEX	3
SETPTR	1	SETCONDPTR	2

Table 1: The number of hardware register accesses (in DWORDS) for each reconfiguration primitive. Allocation and deallocation primitives as measured only operate on metadata (i.e., PDT and PST), not including SRAM/TCAM/action memory resources. The cost for the latter varies depending on the allocation/deallocation sizes.

8.2 Consistency algorithms

Synthesized programs. We evaluate the scalability of FlexCore in generating reconfiguration scripts for programs of different sizes. We generated 100 programs of each size (800 in total), and set $\alpha = 40\%$ for FlexCore to generate reconfiguration scripts. Figure 10a shows the results. As expected, program consistency took the least amount of time, as the only analysis is on the program diff; all edits are then grouped as a

whole. The turnaround time for element consistency grows roughly quadratically with regard to the program size (more strictly, to the size of the diff, which is fixed to 40% of the program size). Execution consistency algorithm lies in between, as it scales with the number of Xch nodes, which is smaller than the program diff. Overall, FlexCore generated reconfiguration scripts for all programs within one second.

Next, we measure the number of invocations of the partial reconfiguration primitives as well as the version metadata operations. As shown in Figure 10b, the numbers of operations for different consistency levels are roughly the same. This is because the number of reconfiguration operations are the same regardless of the consistency level. But the number of transactions increases for weaker guarantees due to the extra version metadata operations.

Real-world programs. We then tested FlexCore on three real-world programs of different sizes, and further varied the reconfiguration ratio α from 5% to 50%. As Figure 9 shows, the FlexCore turnaround time is longer for larger programs and higher reconfiguration ratios. But the overall takeaways are similar as before: FlexCore algorithms scale well for computing reconfiguration scripts. In the Appendix, we further include scalability results for ordering the transactions.

Consistency levels. Figure 11a shows the CDF of the transaction sizes under different consistency guarantees for the synthetic programs with different sizes and α. Under stronger consistency guarantees, the transactions have larger sizes (we fix all tables to the same size). We also measure the headroom requirements. Figure 11b visualizes the step-by-step reconfiguration for one such program: program consistency requires a large peak headroom, but weaker guarantees have less stringent requirements. All consistency levels eventually converge to the same utilization level after reconfiguration completes. Figure 11c tests another program in the software simulator, which plots the percentage of traffic that experiences the old program after the first update is enabled during the reconfiguration under different consistency levels. As we can see, program consistency does not expose any intermediate state, but weaker guarantees lead to more traffic that is

(a) Transaction sizes   (b) Resource headroom   (c) Consistency
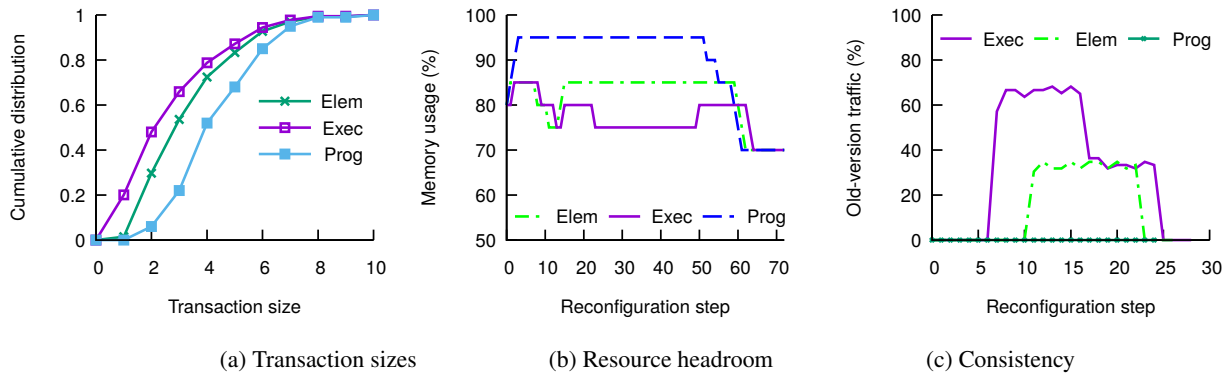
Figure 11: (a) Weaker consistency guarantees lead to smaller transaction sizes; they require lower resource headroom, but more traffic will encounter the old program during reconfiguration. In (b), the initial switch utilization when the reconfiguration starts is 80%. In (c), we use a program whose control flow graph is presented in Figure 5; Y-axis shows the traffic ratio processed by the old program after the first reconfiguration transaction is committed at the 6th step.

processed using the old program during the reconfiguration.

8.3 Case study: Accelerated multicast

Just-in-time optimization. Next, we present a case study using the hardware ASIC, where program elements are injected to the switch pipeline at runtime to accelerate multicast applications. Initially, the switch is configured with a baseline program without any multicast optimizations, and it connects one ZeroMQ unicast sender and multiple receivers. Just before the ZeroMQ application starts, we initiate a partial reconfiguration to extend the switch program with Elmo [34], a switch-based multicast function. Elmo performs source-routed multicast in hardware with customized protocol headers to improve scalability. After ZeroMQ finishes, another reconfiguration removes the Elmo components from the switch pipeline. These changes are performed under program consistency. Each reconfiguration took less than 0.5 s to complete in control plane software.

Just-in-time telemetry. Just before removing Elmo, we inject a co-located telemetry application to observe the effect of Elmo removal, by monitoring the average pipeline latency of randomly sampled packets. This telemetry application is unloaded after the removal of Elmo.

Reconfiguration. Figure 12a plots the throughput of a third background iPerf application during the entire reconfiguration. As we can see, the reconfigurations did not cause any service interruption, as the iPerf throughput was stable throughout the experiment; the switch drop counters also showed no packet loss. Figure 12b plots the additional resource usage in terms of PDT memory, PST memory, and table entries during the runtime reconfigurations. The insertion of Elmo caused a resource usage increase, as did the insertion of the telemetry application. But in both cases, the extra resource overheads for PDT and PST are under 200 bytes. Table rules for multicast and telemetry, on the other hand, are the dominant overheads. All resources are released after the program modules are removed from the pipeline.

Performance. Figure 12c shows that the injection of Elmo improves multicast scalability, where we measure the completion time to send 200 k ZeroMQ messages. Before injecting Elmo, a preceding ZeroMQ run via unicast took up to nearly 60 seconds for six receivers; and the completion time grows roughly linearly with the number of receivers. After injecting Elmo, the switch-based multicast scales independently of the number of receivers, finishing at roughly 20 seconds across all tested configurations. The injected telemetry application detected that the pipeline latency experienced a 20 ns decrease after Elmo was removed from the pipeline.

8.4 Case study: Dynamic telemetry upgrade

In-place application upgrade. We perform another ASIC-based case study under execution consistency. The operator modifies the telemetry application discussed earlier to use different flow keys. Initially, the application uses the IPv4 five tuple as the match key and is configured with 30 k entries. Packets of interest are sampled to software for telemetry processing. The operator issues a reconfiguration to modify the match key to the source and destination addresses instead, using the MODTBL primitive. This modification also reduces the resource usage, as entries become smaller.

Reconfiguration. Figure 13 plots the performance of a background iPerf application, which shows stable throughput. The blue area further shows an additional IPv4 test trace that we generated to specifically trigger the telemetry table. The switch counters indicated zero packet loss for iPerf and the IPv4 test traffic. We have set the migration rate to be 3k entries per second, so the modified table was populated in ~10s. The PDT operations at the control plane software took 400 ms.

Utilization. Figure 14 shows the intermediate program sizes using MODTBL, and compares it with the baseline that inserts the new table and then deletes the old. The baseline incurs a resource utilization spike, which occurs when both tables are co-resident in the switch. As the old table is gradually deallocated, the resource usage drops to the size of the new

(a) Hitless reconfigurations.

(b) Resource overheads.

(c) Performance improvements.
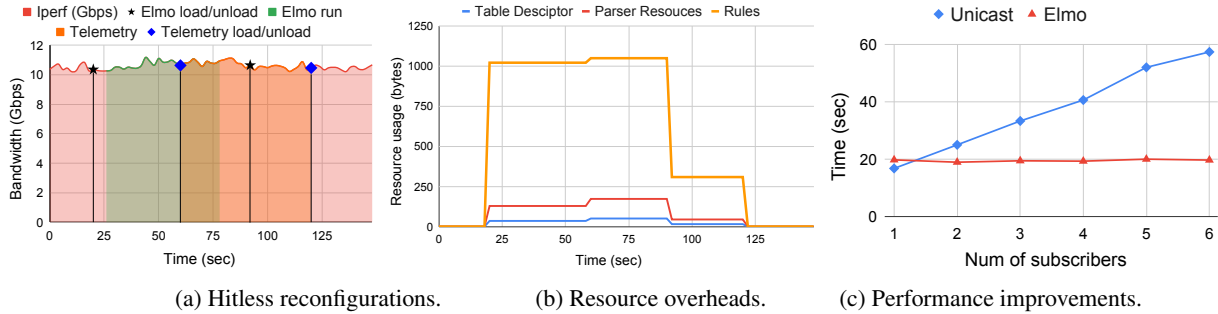
Figure 12: FlexCore inserts Elmo, a switch-based multicast program, just-in-time to accelerate ZeroMQ performance. It also inserts a telemetry application to observe the effect of the removal of Elmo.
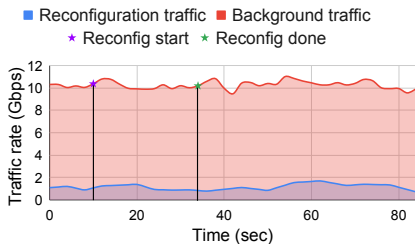
Figure 13: Hitless table modification.
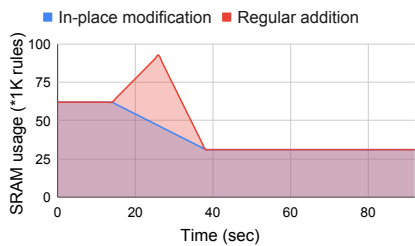
Figure 14: Resource usage with in-place table modification.

table. The final SRAM usage shows the resource reduction in changing the flow keys. In contrast, our in-place modification consistently reduces resource usage reduction right from the beginning, until resource usage approaches its final state.

8.5 Simulator case studies

We have performed two case studies in the software simulator with element consistency. The appendix includes concrete results. We highlight here that all reconfigurations were effective and free of interruptions.

Real-time attack mitigation. This case study injects a *TCP normalizing firewall* [1] and a *covert channel defense* [39] upon attack detection. The normalizer pads all TTL values to avoid inconsistent views at host IDS [22], and the covert channel defense clears TCP reserved bits to avoid data leakage [39]. The normalizer inspects incoming traffic, but the covert channel defense inspects outgoing traffic.

Tenant-specific network extensions. VM migration triggers FlexCore to carve out the tenant's ACL functions from the original switch and inject it to the destination switch.

9   Related work

Programmable networks. Network programmability has been a longstanding goal in the community—starting with 'active networks' [6, 33, 37], each step in this direction has led to significant innovation in the networking ecosystem. FlexCore takes the next step to enable *runtime programmable switches*. Recent projects P4Visor [42] and Hyper4 [16] also use DAG merging algorithms on P4 programs, but our focus is on partial program reconfiguration.

Consistent updates. Network updates are common to datacenters [20, 28], and ensuring the absence of service interrupt is a key goal [24]. Researchers have considered live migration of BPG sessions and virtual routers [21, 38], and per-packet and per-flow consistency guarantees for OpenFlow network updates [29, 30]. Our work tackles the problem of achieving reliable switch program updates at runtime, and proposes a new set of consistency guarantees.

OS+network specialization. The vision of FlexCore is inspired by prior work in OS and network specialization. SPIN [7] is an OS that allows applications to inject safe and dynamic extensions to the kernel. Exokernel [14] enables applications to specialize OS functions at user level. ESwitch [26] specializes OpenFlow software data planes to achieve higher performance for a given workload. Our work aims to enable similar goals for programmable switches.

10   Conclusion

FlexCore argues that runtime programmability should be a first-order goal in future networks, allowing functions to be added or removed dynamically. FlexCore contributes design considerations on switch architectures, partial reconfiguration primitives, reconfiguration algorithms and consistency guarantees. Our evaluation shows that the FlexCore reconfiguration algorithms are scalable, and that runtime reconfigurations are beneficial and free of disruption.

References

[1] Cisco: TCP normalization. `https://www.cisco.com/c/en/us/td/docs/security/asa/asa96/configuration/firewall/asa-96-firewall-config/conns-connlimits.html`.

[2] nplang. `https://github.com/nplang`.

[3] The P4 language repositories. `https://github.com/p4lang`.

[4] The p4c compiler. `https://github.com/p4lang/p4c`.

[5] The P4Runtime Specification. `https://github.com/p4lang/p4runtime`.

[6] D. Scott Alexander, William A. Arbaugh, Michael W. Hicks, Panka J. Kakkar, Angelos D. Keromytis, Jonathan T. Moore, Carl A. Gunter, Scott M. Nettles, and Jonathan M. Smith. The SwitchWare active network architecture. *IEEE Network*, 12(3):29–36, 1998.

[7] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility Safety and Performance in the SPIN Operating System. In *SOSP*, 1995.

[8] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. *ACM SIGCOMM CCR*, 43(4):99–110, 2013.

[9] H. Bunke, X. Jiang, and A. Kandel. On the minimum common supergraph of two graphs. *Computing*, 65(1):13–26, 2020.

[10] Eric Hayden Campbell, William T. Hallahan, Priya Srikumar, Carmelo Cascone, Jed Liu, Vignesh Ramamurthy, Hossein Hojjat, Ruzica Piskac, Robert Soulé, and Nate Foster. Avenir: Managing data plane diversity with control plane synthesis. In *Proc. NSDI*, 2021.

[11] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, et al. dRMT: Disaggregated programmable switching. In *Proc. SIGCOMM*, 2017.

[12] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, et al. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *Proc. NSDI*, 2018.

[13] Dragos Dumitrescu, Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Dataplane equivalence and its applications. In *Proc. NSDI*, 2019.

[14] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An Operating System Architecture for Application-level Resource Management. In *SOSP*, 1995.

[15] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *Proc. SIGCOMM*, 2018.

[16] David Hancock and Jacobus van der Merwe. HyPer4: Using P4 to virtualize the programmable data plane. In *Proc. CoNEXT*, 2016.

[17] Kuo-Feng Hsu, Ryan Beckett, Ang Chen, Jennifer Rexford, Praveen Tammana, and David Walker. Contra: A programmable system for performance-aware routing. In *Proc. NSDI*, 2020.

[18] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soule, Changhoon Kim, and Ion Stoica. NetChain: Scale-free sub-RTT coordination. In *Proc. NSDI*, 2018.

[19] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing key-value stores with fast in-network caching. In *Proc. SOSP*, 2017.

[20] Naga Praveen Katta, Jennifer Rexford, and David Walker. Incremental consistent updates. In *Proc. HotNets*, 2013.

[21] Eric Keller, Jennifer Rexford, and Jacobus E van der Merwe. Seamless BGP migration with router grafting. In *Proc. NSDI*, 2010.

[22] Christian Kreibich, Mark Handley, and V Paxson. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *Proc. USENIX Security*, 2001.

[23] Guanyu Li, Menghao Zhang, Chang Liu, Xiao Kong, Ang Chen, Guofei Gu, and Haixin Duan. NetHCF: Enabling line-rate and adaptive spoofed IP traffic filtering. In *Proc. ICNP*, 2019.

[24] Hongqiang Harry Liu, Xin Wu, Ming Zhang, Lihua Yuan, Roger Wattenhofer, and David Maltz. zUpdate: updating data center networks with zero loss. In *Proc. SIGCOMM*, 2013.

[25] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching ASICs. In *Proc. SIGCOMM*, 2017.

[26] László Molnár, Gergely Pongrácz, Gábor Enyedi, Zoltán Lajos Kis, Levente Csikor, Ferenc Juhász, Attila Kőrösi, and Gábor Rétvári. Dataplane specialization for high-performance OpenFlow software switching. In *Proc. SIGCOMM*, 2016.

[27] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. In *Proc. SIGCOMM*, 2017.

[28] Thanh Dang Nguyen, Marco Chiesa, and Marco Canini. Decentralized consistent updates in SDN. In *Proc.*

SOSR, 2017.

[29] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. In *Proc. SIGCOMM*, 2012.

[30] Mark Reitblatt, Nate Foster, Jennifer Rexford, and David Walker. Consistent updates for software-defined networks: Change you can believe in! In *Proc. HotNets*, 2011.

[31] Jerome H Saltzer, David P Reed, and David D Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 2(4), 1984.

[32] Karla Saur, Joseph Collard, Nate Foster, Arjun Guha, Laurent Vanbever, and Michael Hicks. Safe and flexible controller upgrades for SDNs. In *Proc. SOSR*, 2016.

[33] B. Schwartz, A. W. Jackson, W. T. Strayer, W. Zhou, R. D. Rockwell, and C. Partridge. Smart packets for active networks. In *Proc. OpenArch*, 1999.

[34] Muhammad Shahbaz, Lalith Suresh, Jennifer Rexford, Nick Feamster, Ori Rottenstreich, and Mukesh Hira. Elmo: Source routed multicast for public clouds. In *Proc. SIGCOMM*, 2019.

[35] John Sonchack, Oliver Michel, Adam J Aviv, Eric Keller, and Jonathan M Smith. Scaling hardware accelerated network monitoring to concurrent and dynamic queries with *flow. In *Proc. USENIX ATC*, 2018.

[36] Haoyu Song. Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane. In *Proc. HotSDN*, 2013.

[37] D. L. Tennenhouse and D. J. Wetherall. Towards an active network architecture. *ACM SIGCOMM CCR*, 26(2):5–18, 1996.

[38] Yi Wang, Eric Keller, Brian Biskeborn, Jacobus Van Der Merwe, and Jennifer Rexford. Virtual routers on the move: live router migration as a network-management primitive. *ACM SIGCOMM CCR*, 38(4):231–242, 2008.

[39] Jiarong Xing, Qiao Kang, and Ang Chen. Netwarden: Mitigating network covert channels while preserving performance. In *Proc. USENIX Security*, 2020.

[40] Nofel Yaseen, John Sonchack, and Vincent Liu. Synchronized network snapshots. In *Proc. SIGCOMM*, 2018.

[41] Liangcheng Yu, John Sonchack, and Vincent Liu. Mantis: Reactive programmable switches. In *Proc. SIGCOMM*, 2020.

[42] Peng Zheng, Theophilus Benson, and Chengchen Hu. P4Visor: Lightweight virtualization and composition primitives for building and testing modular programs. In *Proc. CoNEXT*, 2018.

11 Appendix

11.1 Case study: Real-time attack mitigation

In this case study, we present how FlexCore facilitates real-time attack mitigation by reconfiguring two defense functions to the software switch simulator.

Traffic normalizers [22] are firewall utilities that prevent inconsistent views between network IDS and end hosts. As an example, some packets may be seen by the IDS, but their TTL values are crafted in such a way that they are dropped soon after the network IDS and do not trigger processing at end hosts. This leads to vulnerabilities [22]. A normalizer firewall can pad TTL values to ensure that the IDS and the hosts always have the same view.

Covert channels [39] leak secret data by repurposing packet header fields as data carrier. For instance, an attacker that compromise a server that hosts confidential data may leak the secret by padding them into the TCP reserved bits of network traffic. A defense needs to clear such optional header fields to prevent leakage.

Real-time attack mitigation. In our case study, we inject a *TCP normalizing firewall* [1] and a *covert channel defense* [39] upon attack detection. Since these two defenses are independent, they can be reconfigured under element consistency. Figure 15 shows the workflow for the reconfiguration. After each defense is deployed, the attack traffic can be recognized and blocked; its throughput drops to zero. The normal traffic does not experience any loss or interruption during the reconfiguration.

11.2 Case study: Tenant-specific network extensions

In this case study, we focus on multi-tenant datacenters where each tenant can inject her own network extensions to the switch. Upon VM migration, the switch modules are carved out from the source and grafted to the new destination switch.

Program grafting in VM migration. In this scenario, a tenant has her ACL module injected to the ToR switch, and her VM migration will bring this module to a different destination rack. This is achieved by carving out the ACL components and grafting them to the destination switch using partial reconfigurations. Figure 16 shows the traffic rate of the tenant's traffic and the background traffic. The migration is achieved in several steps. It first inserts the ACL module to the new switch, and then routes traffic to the new switch by updating the routing rules of upstream switches. Finally, it removes the ACL module in the old switch. As we can see, the migration does not cause throughput drops of the background traffic during the reconfiguration, and the tenant's traffic is migrated to the new switch without service interruption.

11.3 Evaluation: Ordering the transactions

For element and execution consistency, the reconfiguration proceeds in multiple steps. So FlexCore additionally performs an exhaustive search to identify a feasible sequence under the current headroom. The problem can be stated as: given
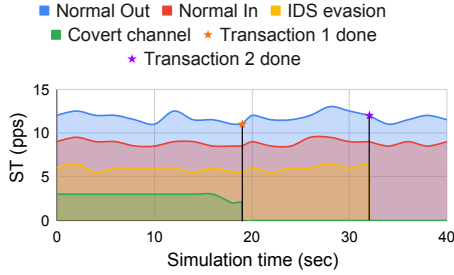
Figure 15: Simulation traffic rates (ST) when reconfiguring the switch using element consistency to inject real-time network defenses.
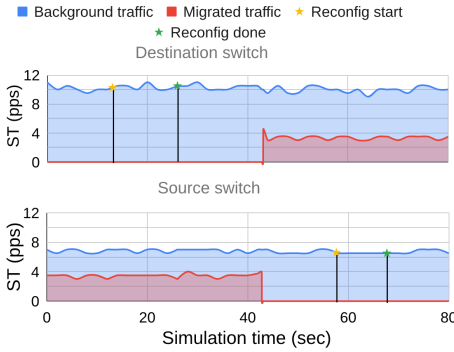
Figure 16: Simulation traffic rates (ST) during a reconfiguration triggered by VM migration, which carves out an tenant-specific ACL module from the source switch and grafts it to the destination switch.

(a) Element consistency   (b) Execution consistency
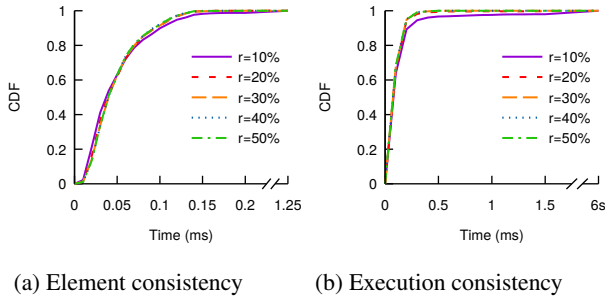
Figure 17: The turnaround time for finding a feasible sequence for the transactions, under element and execution consistency.

a set of transactions $tx_1, tx_2, \cdots, tx_k$, find a feasible sequence that fits into the current resource headroom, or conclude that such a sequence doesn't exist. For element consistency, the transactions do not have a hard constraint as to their order. The search only focuses on optimizing for resource headroom. Execution consistency has hard constraints as to which transactions should be ordered before others. The search also encodes such constraints as induced from the XCH nodes.

In Section 8.2, we have evaluated the scalability of the algorithm in generating reconfiguration scripts. Here, we further evaluate the turnaround time for identifying a feasible sequence of transactions that can be applied within the available headroom. We have used an α ranging from 10% to 50% on the synthesized programs, and tried different switch resource

headrooms as denoted by r (ranging from 10% to 50%).

Figure 17a shows the results for element consistency. We can see that FlexCore finishes within 0.2ms for all programs across different headrooms except for r=10%. With 10% headroom, the switch has very small slack for the reconfiguration, so it takes more time to search for a feasible plan or determine that no solution exists.

Figure 17b shows the results for execution consistency, where the turnaround time is higher because of two reasons. First, execution consistency needs to merge and deduplicate the subpartitions following their constraints. Second, execution consistency could generate more candidate solutions, resulting in longer searching time. However, the algorithm can still complete within 1.5ms for 98% programs and within 6s for all programs.

IMap: Fast and Scalable In-Network Scanning with Programmable Switches

Guanyu Li[*], Menghao Zhang[*†], Cheng Guo[*], Han Bao[*], Mingwei Xu[*], Hongxin Hu[°], Fenghua Li[*]
[*]*Tsinghua University*   [†]*Kuaishou Technology*   [°]*University at Buffalo, SUNY*

Abstract

Network scanning has been a standard measurement technique to understand a network's security situations, e.g., revealing security vulnerabilities, monitoring service deployments. However, probing a large-scale scanning space with existing network scanners is both difficult and slow, since they are all implemented on commodity servers and deployed at the network edge. To address this, we introduce IMap, a fast and scalable in-network scanner based on programmable switches. In designing IMap, we overcome key restrictions posed by computation models and memory resources of programmable switches, and devise numerous techniques and optimizations, including an address-random and rate-adaptive probe packet generation mechanism, and a correct and efficient response packet processing scheme, to turn a switch into a practical high-speed network scanner. We implement an open-source prototype of IMap, and evaluate it with extensive testbed experiments and real-world deployments in our campus network. Evaluation results show that even with one switch port enabled, IMap can survey all ports of our campus network (i.e., a total of up to 25 billion scanning space) in 8 minutes. This demonstrates a nearly 4 times faster scanning speed and 1.5 times higher scanning accuracy than the state of the art, which shows that IMap has great potentials to be the next-generation terabit network scanner with all switch ports enabled. Leveraging IMap, we also discover several potential security threats in our campus network, and report them to our network administrators responsibly.

1 Introduction

Network scanning is a typical procedure to discover active hosts, ports and services in a network, which is mainly used by network operators/researchers for security assessment and system maintenance of the network. Enabled by tools such as Nmap [39], ZMap [14] and Masscan [33], network scanning has become a standard measurement technique to understand host behaviors in the target network, even the entire Internet.

Recent studies have demonstrated that network scanning can help reveal new security vulnerabilities [3, 6, 10], monitor service deployments [2, 13, 20, 42] and shed light on previously opaque distributed systems [19], which are essential for people to understand the network's security situations.

Today's network scanners, however, cannot keep pace with today's soaring scanning space and provide a timely security snapshot. Recently IPv6 has proceeded to the stage of large-scale deployment, and reports show that IPv6 is used by 18.7% of all the websites [47]. Along with the adoption of 5G networks, more and more Internet-of-Things (IoT) devices and mobile devices are connecting online [4]. The increased address space and the numerous online devices mean that the network scanner should be *scalable* to this much larger scanning space easily. Moreover, since these IoT and mobile devices go online and offline frequently, it is necessary for network scanners to conduct a comprehensive scanning quickly. Otherwise, a large number of security snapshots cannot be captured, potentially missing numerous security incidents [46]. This raises the requirement that the network scanner should complete a comprehensive scanning as *fast* as possible.

However, a closer look into today's network scanners shows that they are far from being fast and scalable due to their implementation targets and deployment locations. First, in terms of implementation targets, current network scanners are all implemented on commodity servers. As CPUs on servers are not specialized for high-speed packet processing, the scanning speed of these CPU-based network scanners is intrinsically limited. Second, in terms of the deployment locations, state-of-the-art network scanners are all located at the network edge. Scanning from the edge is usually limited by the upstream bandwidth of the end host, which inevitably constrains the utmost scanning speed for network scanning tasks. Besides, the end-to-end scanning paths indicate more bandwidth waste for edge networks and larger possibilities of dropping probe/response packets.

In this paper, we propose IMap, a fast and scalable in-network scanner to address the aforementioned issues. The technology enabler for IMap is the emergence of pro-

grammable switches [9], which offer unprecedented programmability and flexibility without sacrificing performance. Generally speaking, one single programmable switch could provide a packet processing capability as high as multiple Tbps, which is several orders of magnitude higher than highly-optimized servers. Besides, such switches support stateful packet processing with domain-specific languages (e.g., P4 [8]), which allows programmers to enforce user-defined packet processing logics in the switch pipeline directly. Moreover, switches (especially core switches) provide a unique vantage point for network scanning, which is no longer constrained by the upstream bandwidth of the end host or plagued by the bandwidth waste of the end-to-end scanning paths. These unique characteristics of programmable switches are incredibly valuable for the next-generation fast and scalable network scanners.

Nevertheless, designing IMap is a non-trivial effort. As an in-network scanner, when sending probe packets, IMap must cover the scanning space completely, and also be aware of network conditions to avoid affecting the normal packet routing functionality. Besides, once response packets arrive, IMap should distinguish normal packets and response packets correctly, and also process the response packets efficiently to avoid saturating the storage server. However, switches only have constrained computational models and limited memory resources, which cannot satisfy these requirements easily.

To meet these requirements, IMap designs a set of techniques and optimizations, i.e., an address-random and rate-adaptive probe packet generation mechanism, and a correct and efficient response packet processing scheme, to turn a switch into a high-speed network scanner. We implement a prototype of IMap in an Intel Tofino switch [23], and make the source code publicly available [22]. Testbed experiments and real-world deployments show that even with one switch port enabled, IMap can survey all ports of our campus network (i.e., 6 Class B IP Addresses), a total of up to 25 billion scanning space, in 8 minutes, achieving a nearly 4 times faster scanning speed and 1.5 times higher scanning accuracy than state-of-the-art network scanners. IMap also discovers several potential security threats in our campus network. To the best of our knowledge, IMap is the first network scanner that can potentially reach multiple Tbps scanning speed, benefiting from its implementation targets and deployment locations. We hope IMap can serve as the foundation for next-generation terabit network scanners.

In summary, we make following contributions in this paper:

- We analyze the limitations of current network scanners, and identify the opportunities brought by programmable switches (§2).

- We propose IMap, a fast and scalable in-network scanner with programmable switches. IMap consists of a probe packet generation module to generate high-speed probe packets with random address and adaptive rate, and a

response packet processing module to process response packets correctly and efficiently (§3, §4).

- We implement an open-source prototype of IMap, and conduct extensive testbed experiments and campus network deployments to show advantages of IMap (§5, §6).

Finally, we make some discussions in §7, describe related works in §8, and conclude this paper in §9.

2 Motivation and Observation

2.1 Limitations of Current Network Scanners

With the rapid growth of scanning spaces and security incidents recently, today's network scanners are falling behind the times, especially in terms of scanning *scalability* and scanning *speed*. First, network scanners should be able to scale to large scanning spaces easily. Recently IPv6 has been in the stage of large-scale adoption, for instance, Google's statistics show that around 35% of its users access Google via IPv6 [24]. Since IPv6 has a much larger address space than IPv4, the scanning space increases drastically. Besides, along with the deployment of 5G networks, more and more IoT/mobile devices are connecting online [4]. All these require that network scanners should be able to cover a large scanning space easily. Second, network scanners should be fast enough to provide timely security snapshots. Today's networks become more and more dynamic, and IoT/mobile devices switch between online and offline frequently. Meanwhile, we have also witnessed that security incidents occur more and more frequently, and some of them occur in a very small time scale (e.g., from tens of seconds to several minutes). For example, according to Cybint's monthly newsletter, since COVID-19, the frequency of cybercrimes increases 300%, and hackers attempt to attack vulnerable home networks as people are working from home [46]. As a consequence, network scanners should be able to complete a comprehensive scanning as fast as possible. Otherwise, some security snapshots cannot be captured and important security incidents may be missed.

However, today's network scanners are intrinsically slow, which are far from being fast and scalable to satisfy the aforementioned new requirements. For example, with Zippier ZMap [1], one of the most powerful network scanners today, the scanning capability only reaches a throughput of 10 Gbps and a rate of 14.2 Mpps [45]. The capability of today's network scanners is limited by two key factors fundamentally. First, in terms of implementation targets, current network scanners are all implemented on commodity servers. Packet processing on commodity servers is intrinsically slow, since CPUs are not specialized for high-speed packet processing. Even with software optimizations like DPDK [12], the throughput cannot reach more than 40 Gbps easily [25,41,50]. Second, in terms of deployment locations, today's network scanners are all located at the edge of the network. Scanning

from the edge is not only limited by the upstream bandwidth of the end host, but also incurs longer scanning paths and non-negligible bandwidth waste because of end-to-end scanning paths. As a result, even the scanners are capable of scanning at higher rate (e.g., 40 Gbps), the scanning results (e.g., hit rate, active/inactive rate) may suffer from low accuracy because of undesirable probe/response packet drops on the end-to-end scanning paths (§6.2). Not surprisingly, because of these fundamental limitations, since the publication of Zipper ZMap [1], the network scanning tools have not experienced any progress, and researchers have turned to improve the scanning accuracy with the help of various algorithmic techniques [7, 15, 16, 21, 36].

2.2 Opportunities by Programmable Switches

Programmable switches [8, 9] bring unprecedented opportunities to address the limitations of current network scanners.
High packet processing capability. Switching ASICs are specialized for high-speed line-rate packet processing, which can provide several orders of magnitude higher throughput than highly-optimized servers [25]. Specifically, today's latest CPU-based network scanner, Zipper ZMap [1], could only provide a scanning rate of 14.2 Mpps and a scanning throughput of 10 Gbps. In contrast, switching ASICs can easily process a few billion packets per second, which shows great potentials to be a terabit network scanner. Other hardware alternatives, such as FPGA and NPU, cannot match the performance of switching ASICs [25], thus not promising for a high-speed network scanner.
Flexibility to support scanning tasks. The most prominent characteristic of the new-generation switching ASICs is programmability. Such switching ASICs can be programmed with domain-specific languages like P4 [8], and also support stateful packet processing with user-defined logics. Besides, programs can run collaboratively between the data plane switching ASICs and the control plane switch CPUs, enabling advanced and flexible packet processing. As a result, diverse scanning tasks can be implemented in the programmable switch, which would potentially be the foundation of next-generation high-speed network scanners.
Vantage points to conduct network scanning. Existing network scanners are all located at network edges and implemented in end hosts, where the utmost scanning rate is usually constrained by the bandwidth of the end hosts. Worse yet, scanning from the end host requires an end-to-end scanning path, which inevitably results in the waste of bandwidth resources and the degradation of scanning accuracy. In contrast, switches provide a unique vantage point for network scanning tasks. Core switches usually have huge spare bandwidths (i.e., more than 50% spare bandwidth [11]), which shows substantial potentials for network scanners to tap. Moreover, scanning from a core switch is no longer plagued by the bandwidth waste or the scanning accuracy degradation resulted from
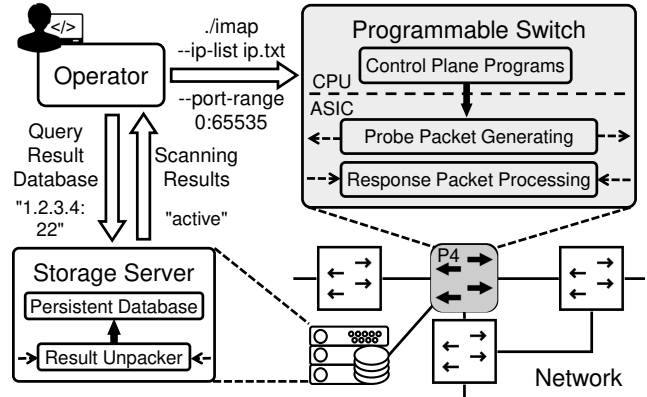
Figure 1: The workflow of IMap.

the end-to-end scanning path. This scanning vantage point is particularly valuable for high-speed network scanners.

3 IMap Overview

3.1 Deployment Scenario

Our scenario focuses on a network-centric deployment model, where the administrators of an ISP or a cloud network deploy IMap to understand their own network's security situations. IMap could also be used for Internet-wide scanning, but this should avoid causing any ethical concerns, as pointed out in ZMap [14]. Ideally, IMap should be built on a core switch, which provides both routing services and scanning functionality simultaneously. In other words, the IMap switch should first preserve the functionality of packet switching, and then behave as a high-speed network scanner when there is spare bandwidth (e.g., reports show that bandwidth occupation ratio for core switches is usually less than 50% [11]). Note that the deployment of programmable switches is not a new requirement; several ISPs/cloud networks have already replaced their legacy switches with programmable switches in their networks, which we believe is an irresistible trend in the foreseeable future [40, 43, 44]. Besides, an in-core-network scanner also raises the bar for attackers to take advantage of this powerful network scanner, as it is difficult for normal attackers to obtain such a deployment location.

3.2 Workflow and Design Requirements

IMap is designed to be a high-speed, easy-to-use network scanner, so the usage of IMap is similar to traditional network scanners, such as ZMap [14] and Masscan [33]. As shown in Figure 1, operators should first specify the scanning address spaces and scanning port ranges beforehand. Then IMap control plane programs parse these configurations and issue the parsed parameters into the IMap packet processing logic. After that, IMap data plane programs generate high-speed probe packets and process response packets accordingly. Finally, the

scanning results, i.e., the information extracted from the response packets, are written into a persistent database, such as a Redis in-memory data store [29]. In the design, implementation and deployment of IMap, we identify several different design requirements that must be satisfied to make IMap a practical high-speed network scanner, especially in terms of probing packet generation and response packet processing:

Space-complete and rate-adaptive probe packet generation(§4.1). In terms of probe packet generation, there are two key requirements in switch-based high-speed scanning. First, IMap should be able to cover the desired scanning space (i.e., |address space| × |port space|) completely, without duplications and omissions. This is a basic functional requirement for a network scanner. Second, packet switching is the first-class citizen of the switch, therefore, IMap should be able to conduct network scanning tasks without affecting normal network routing functionality. As the spare bandwidth of the network is dynamic, we need a network-aware method to generate high-speed probe packets with adaptive rate.

Correct and efficient response packet processing(§4.2). With regards to response packet processing, we also have to fulfill two requirements. First, switches are also responsible for normal packet forwarding, therefore, the input packets for the switch-based scanner have both normal packets and response packets. As a result, the scanner should be able to distinguish normal packets and response packets correctly. Second, response packets cannot be steered to servers directly, as it may saturate the bandwidth of the storage servers and overwhelm the writing capability of the database. The scanner should have an efficient response packet processing approach to reduce the server-side pressure.

4 IMap Design

4.1 Probe Packet Generation

Switch is designed to be a packet forwarding device, not a packet generation device, thus cannot generate probe packets without ground. Inspired by HyperTester [49], we also leverage the template-based packet generation mechanism to generate high-speed probe packets. As shown in Figure 2, the switch CPU first prepares a set of template packets with initialized headers, and injects them into switching ASICs. Our tests manifest 50k template packets are enough for line-rate scanning and the injection takes 15 ms, causing negligible loads on the switch CPU. After receiving these template packets, switching ASICs keep looping these packets in the switch pipeline, where each packet experiences three sequential steps: an *accelerator* to accelerate the template packets to 100 Gbps line rate, a *replicator* to replicate the template packets into several switch ports, and an *editor* to edit the headers of replicated template packets into desired probe packets.

(1) **Accelerator.** The accelerator is located at the ingress pipeline, and it keeps looping the template packets by inject-
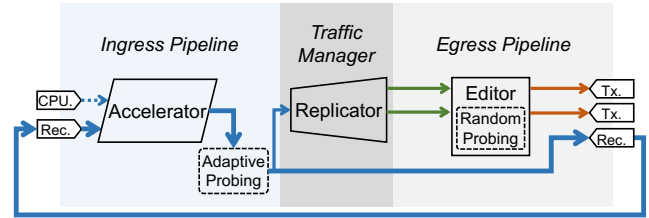
Figure 2: Probe packet generation of IMap.

ing these packets into the *recirculate port*. The recirculate port is a special port in the switch pipeline, where the injected packets are sent back to the ingress pipeline immediately. Therefore, after injecting a set of template packets to fill the switch pipeline, we get a 100 Gbps line-rate stable packet source for the replicator.

(2) **Replicator.** The replicator is located at the traffic manager, which mainly takes the template packets from the accelerator as input and replicates these packets into a given port set with the *packet replication engine*. The packet replication engine is a hardware component in the traffic manager, which is widely supported by today's programmable switches. By configuring a set of ports for multicast from the control plane, incoming packets will be replicated and forwarded to the given port set in parallel. The original template packets from the accelerator will continue to be recirculated across the switch pipeline, to ensure line-rate stable packet source for the replicator, and the replicated template packets would go through the editor for further processing.

(3) **Editor.** The editor resides in the egress pipeline, and it is responsible to modify the replicated template packets into the desired probe packets. As long as the packet headers can be parsed by programmable switches, the headers can be set to given values, e.g., constants, or values from registers. To turn replicated template packets into probe packets, some header fields (e.g., destination IP address, destination port) need modification via the editor, while other fields (e.g., protocol type, source IP address) are inherited from the template packets which are created by the switch CPU initially.

With the steps above, we obtain continuous probe packets at line rate in multiple egress ports. Nevertheless, to be a practical high-speed network scanner, IMap should be able to generate probe packets to cover the scanning space (i.e., |address space| × |port space|) completely, and adapt the scanning rate according to the network conditions.

4.1.1 Random probe address

To cover the scanning address space completely, an intuitive way is to scan from the start IP address to the end IP address one by one. Nevertheless, simply probing IP addresses in numerical order would overwhelm the target networks with the scanning traffic, which may produce inconsistent probing results and incur complaints from the target networks. To avoid this, IMap should be able to scan the addresses accord-
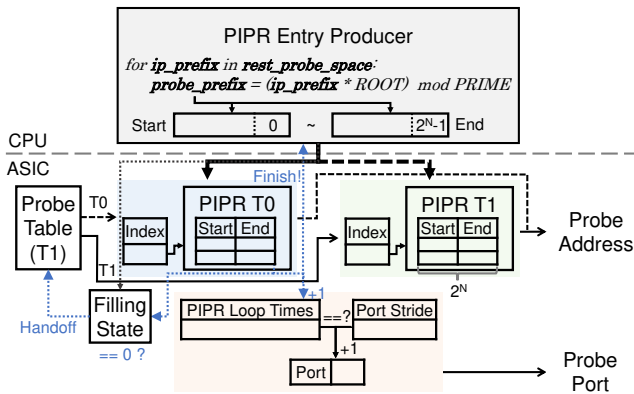
Figure 3: Random probe address.

ing to a permutation of the address space, without duplications and omissions. However, the switching ASICs only have limited programmability and memory resources, which cannot support complex calculations or maintain massive states. The address generation approach in ZMap [14] requires calculations such as multiplication and modulo, thus is not feasible in the switching ASICs.

To address this problem, we leverage the flexibility of the switch CPU to supplement the switching ASICs to generate line-rate address-random probe packets. In the editor of the switching ASICs, we design a Probe IP Range (PIPR) table based on register arrays. In the switch CPUs, we have a PIPR Entry Producer module. Using the address generation method similar to ZMap [14], the PIPR Entry Producer module can generate a random permutation of the probe IP ranges for a given address space. After the PIPR Entry Producer module fills part of the generated probe ip ranges into the PIPR table, probe packets can iterate through the PIPR table to obtain the random destination IP addresses. As the data plane scanning is pretty fast, a PIPR table with entry size of 1 will be scanned quickly, so we store a probe ip range in each entry of the PIPR table. To implement this, our PIPR table consists of two register arrays: one is named as PIPR_Start array, which is used to store the start of the probe ip range; the other is named as PIPR_End array, to store the end of the probe ip range. Before the PIPR table, we have a PIPR_Index register, which is used to index the PIPR table. The initial value of the PIPR_Index register is set as 0 by the control plane; upon an incoming probe packet, the value of PIPR_Index increases by 1, until the size of the PIPR table; after that, the PIPR_Index is assigned as 0 again and another loop starts. For the PIPR_Start array, upon each incoming packet, the corresponding PIPR_Start register increases by 1, until the PIPR_End register. When the value of the last PIPR_Start register is equal to the value of the last PIPR_End register, the scanning for the current PIPR table is finished, and the PIPR Entry Producer module is supposed to fill a new round of probe ip ranges into the PIPR table. To send the finish signal to the control plane, we leverage the *egress to egress mirror* primitive in the switch pipeline, which can carry a predefined flag to the switch CPU

to notify the PIPR Entry Producer module.

However, conducting a new round of PIPR table filling is a time-consuming task. According to our tests on the Intel Tofino switch [23], even with the batching optimization, filling a PIPR table with size of 65,536 requires about 0.3 seconds. This indicates that, after a round of scanning, the data plane has to wait for at least 0.3 seconds to start the next round of scanning. This is unacceptable for high-speed scanning, as it compromises the scanning rate significantly. To resolve this problem, we introduce two PIPR tables and PIPR_Index registers. When one PIPR table is being scanned, the other PIPR table is being filled with the next round of probe ip ranges. To make the two PIPR tables handoff seamlessly, we design a Probe_Table register in the first stage of the egress pipeline, which is switched between 0 and 1, and controls the flow of probe packets. The switching of the Probe_Table register is triggered by the finish signal of the egress to egress mirror primitive. Definitely, to achieve continuous probe packets, there is a mathematical relation that the PIPR table size, the PIPR table filling time, and the scanning rate must satisfy. Supposing the size of the PIPR table is N, the difference between each PIPR_Start register and PIPR_End register (i.e., the size of a PIPR table entry) is L, the PIPR table filling time is T seconds, the total scanning rate R (packets per second) should satisfy that $R \leq \frac{N \times L}{T}$. However, there are still a few extreme scenarios where the actual PIPR table filling time is longer than the expected T, e.g., caused by the congestion of the switch CPU or the control channel. It means the inequality is not held and the PIPR table is being read before fully filled. To deal with such cases, we add a Filling_State register before the PIPR_Table register to indicate whether the PIPR table filling is finished. It is set to 1 when the control plane begins to fill and set to 0 when the control plane finishes the filling. The finish signal of the egress to egress mirror primitive will check whether the Filling_State register is 0 before it switches the PIPR_Table register.

Until now, the designs above only consider one port scenario, which should be extended to support a port range scenario, e.g., scanning from port 22 to port 80. Since the scanning address already has good randomness, we choose to scan the port one by one. However, updating the Port register from the control plane would bring about race conditions, as the high-speed probe packets are already looping in the switch pipeline. To address this, we design a port self-increment mechanism in the data plane. As the control plane knows in advance the number of times the scanning address space needs to loop in the PIPR table, we design a Port_Stride register in the switch pipeline, which is filled with the number of loop times by the control plane. Every time the scanning of one PIPR table finishes, the corresponding counter increases by 1, until the value of the Port_Stride register. Then, the Port register increases by 1 and the counter is set as 0 again. With all the mechanisms above, the final design of our random probe address is described in Figure 3, which achieves to generate

address-random probing packets to cover the scanning space completely, without overwhelming target networks.

4.1.2 Adaptive probe rate

To avoid affecting the normal packet routing functionality of the network, IMap desires a network-aware method to generate high-speed probe packets with adaptive rate. The "adaptive" here has two kinds of meanings. First, the control plane of the IMap switch should be aware of the nearby network conditions for further scanning rate adjustment. Furthermore, the IMap data plane should have a rate-adjusting interface, which can receive commands from the control plane to accurately adjust the scanning rate.

To be control plane aware, IMap should be able to adjust the scanning rate according to the network conditions. We formulate the scanning rate adjustment problem as follows. The scanning network can be modeled to a graph $G = (V, E)$, where V and E are sets of forwarding devices and directed links between devices. Note that link $e = (v_i, v_j)$ is directed, and (v_i, v_j) and (v_j, v_i) are different links. Each link $e \in E$ has a capacity c_e and its current load is represented with l_e. We assume there exists a monitoring system in the network, so l_e can be obtained with the port bandwidth usage of the devices connected by e periodically. IMap is deployed in $v_{IMap} \in V$ and its ports $P_{IMap} = \{p\}$ connect to the network with links $\{e_p\} \subset E$. The maximal scanning rate for port p is c_{e_p}, which is the bandwidth capacity of the link e_p. According to the routing table on v_{IMap}, we can partition the scanning space S by P_{IMap} in advance so that each port p corresponds to a routing-aware sub-scanning space $s_p \subset S$. Besides, we can estimate the extra load $d_{p,e}$ on each link e caused by full-rate probe packets of s_p. This can be done by configuring IMap to send probe packets of s_p with a specific tag on port p at low rate, then using the monitoring system to detect the load caused by the traffic with the given tag, and finally inferring $d_{p,e}$ when the scanning rate is c_{e_p} [28, 31]. Such partition and estimation should be repeated to adapt to routing dynamics when the routing tables in the scanning network change drastically. Then the scanning rate adjustment problem can be solved based on the Linear Programming (LP), as follows:

$$max \qquad \sum_{p \in P_{IMap}} \alpha_p c_{e_p} \qquad (1)$$

$$s.t. \forall e \in E: \ l_e + \sum_{p \in P_{IMap}} \alpha_p d_{p,e} \leq \beta c_e \qquad (2)$$

where $0 \leq \alpha_p \leq 1$ denotes the rate throttling parameter and $0 \leq \beta \leq 1$ denotes the maximum bandwidth occupation ratio. α_p is the output of this formulation and β is set by administrators to make the network robust for burst traffic. Equation (1) indicates that the objective is to maximize the total scanning rate on all ports. And Equation (2) states the extra load brought by IMap can not overwhelm any link in the network. Given $\{\alpha_p\}$, the control plane can determine the scanning rate

for each port. Note that our current design fits for one single Autonomous System (AS) network; for inter-AS networks, as different networks belong to different administrative domains and they are not willing to share confidential information (e.g., network topology, network utilization), it is extremely difficult to design an inter-AS network-aware rate adjustment approach accurately. IMap is mainly designed for the single-AS network scanning, and only provides a best-effort probing service for inter-AS network scanning tasks.

To make the scanning rate of IMap adjustable, we add a *throttle* in the switch pipeline, which can be adjusted from the control plane dynamically. Located in the ingress pipeline, the throttle is used to determine when the replicator could replicate the template packets. In general, the switching ASICs can provide a per-port 100 Gbps packet processing capability, thus enabling nanosecond-level (e.g., ∼6 nanoseconds for 64-byte packets) timestamp for each incoming packet. Our throttle consists of two registers in the switch pipeline. The first one is named as a timestamp register, which is used to record the timestamp of the last template packet that is successfully replicated and sent out to the editor. For every incoming template packet, we calculate the difference between the timestamp of the current packet and the timestamp recorded in the timestamp register. Upon the difference exceeds a certain threshold, we pass the template packet to the replicator and update the recorded timestamp. The second one is named as a rate register, which is used to make the aforementioned threshold configurable from the control plane. In the ingress pipeline, the rate register resides in the front of the timestamp register, and the control plane programs can fill the certain value into the rate register to achieve the rate control.

4.2 Response Packet Processing

As an in-network scanner based on the core switch, IMap is also responsible for forwarding normal packets, e.g., packets from other routers and switches in the network. IMap should be able to distinguish normal packets and response packets correctly. Meanwhile, since the throughput of response packets may be large, IMap should be able to efficiently process the response packets to avoid saturating the storage server.

4.2.1 Distinguishing normal/response packets

To distinguish response packets from normal packets, one approach is to maintain a secret state for each probe packet, and then verify whether the response packet is corresponding to the secret state accordingly. However, the switching ASICs only have limited memory resources, which cannot maintain massive secret states.

To resolve this, we design a stateless connection mechanism similar to SYN cookies [5]. Rather than maintaining states in the switching ASICs, we encode the secret state into the mutable fields of each probe packet. The fields should
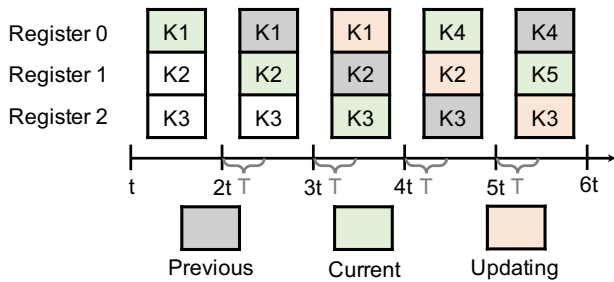
Figure 4: Key update procedure of IMap.

Figure 5: Response packets aggregating in IMap.

have recognizable effects on fields of the corresponding response packets. Specifically, for TCP scanning, we choose the source port and initial sequence number; for ICMP, we use the ICMP identifier and sequence number; for UDP, we use the source port. Take TCP as a concrete example, in the egress pipeline, when IMap sends a probe packet, the editor sets *SrcPort* as $hash(Key, Proto, SrcIP, DstIP)$, and *SeqNo* as $hash(Key, Proto, SrcIP, DstIP, SrcPort, DstPort)$, where *Key* is a secret key maintained in the register of the switching ASICs. Accordingly, in the ingress pipeline, IMap has a *verifier*, which checks the *DstPort* and *AckNo* to determine whether the received packet is a valid response to the probe packet. ICMP scanning and UDP scanning work in a similar manner, except for different packet fields. After the verifier checks the validation of the response packets, similar to ZMap [14], IMap also responds a TCP RST packet to each SYN-ACK packet to close the TCP connection.

One potential issue with the method above is the security of the verifier. Currently the hash functions supported in the switching ASICs (e.g., CRC32) are relatively simple, which are not true cryptographic functions and are vulnerable to *chosen plaintext attacks* [48]. As a result, attackers may perform such attacks to restore the *Key*, and deliberately inject forged response packets to pollute the scanning results. To further enhance the security of the verifier and enable pollution-free scanning results, IMap updates the *Key* every *t* seconds. This can reduce the damage caused by compromised secret keys to a large extent: even if an attacker somehow manages to obtain the current key, such knowledge will become useless after at most *t* seconds.

However, simply updating the *Key* would result in inconsistent scanning results. For example, *Key*1 is updated to *Key*2 after IMap sends the probe packet. Soon the response packet arrives, the verifier determines this packet is invalid as the current key cannot obtain a correct validation for its packet headers. To address the inconsistency issue described above, IMap stores the last key used for a certain period of time. More specifically, IMap maintains three keys (i.e., the previous key, the current key, and the next key) at any given time. Every *t* seconds, IMap rotates a slot index from 0 to 2, and the key in $slot_i$ is used for the hash function. Each key can stay in a slot for at most 3*t* seconds; after 3*t* seconds, the key is updated by the control plane. A concrete example is shown in
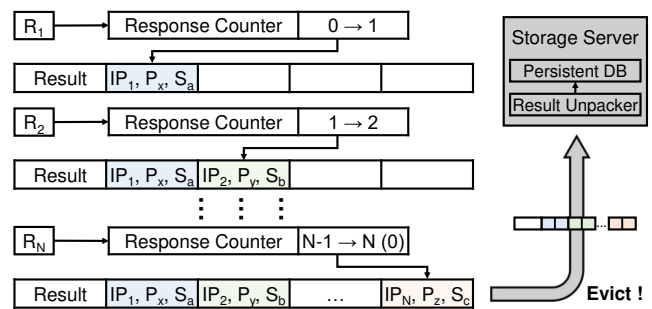
Figure 4, where *T* denotes the max time interval between any probe packet and the corresponding response packet. The editor will encode the 2-bit slot index of the key into the header fields of the probe packet, and the fields should also be added in the corresponding response packet within this connection. Currently, we encode it into the source port for TCP/UDP and the identifier for ICMP. Based on the slot index, the verifier can conduct the validation correctly.

4.2.2 Aggregating response packets

To avoid saturating the storage server, IMap desires an efficient response packet processing approach. One intuitive approach is to use hash mechanisms [17, 35, 49]. However, as the key set is really large in IMap (e.g., the size of the scanning address space), even only storing 2-bit value for each key requires GB-level memory, which exceeds the memory resources of the switching ASICs (i.e., 50-100MB [35]) significantly.

To resolve this problem, instead of seeking to store all the keys/values, we adopt a response packet aggregation mechanism that is compatible with the current switching ASICs. More specially, as shown in Figure 5, IMap designs a dedicated *N*-size register array to temporarily store the scanning results. For each incoming response packet, IMap extracts its source IP, source port and state (i.e., active or inactive), and records the information in one register. When the register array is filled up, the corresponding response packet packs all the results from the register array, and goes to the storage server. To determine which register stores which result, we implement a counter in the ingress pipeline. Upon an incoming response packet, the counter increases by 1. The information extracted from the *i*-th packet will be stored in the *i*-th register. The *N*-th response packet will trigger the replication and be sent to the switch port connected with the storage server, packing and carrying all the results from the register array. Meanwhile, the counter is reset as 0 and another aggregation loop starts. With this approach, IMap achieves an *N* to 1 aggregation, reducing the pressure for the bandwidth of the storage server significantly. In the side of the storage server, we use DPDK [12], a high-performance I/O framework, to parse the result packets and extract the scanning results. Finally, the scanning results are stored in a persistent database.
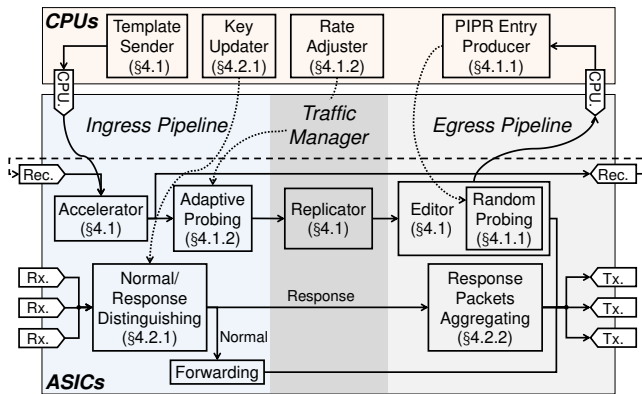
Figure 6: Component layout of IMap.

5 Implementation

We implement a prototype of IMap, and make our code publicly available here [22]. Figure 6 illustrates the component layout of IMap on the data plane switching ASICs and the control plane switch CPUs.

The data plane part is implemented with ∼2K lines of P4-16 code for the Intel Tofino ASIC. In the probe packet generation module, we set the size of PIPR tables as 65536 and the size of one PIPR table entry as 256. In the response packet processing module, we utilize CRC32 as the *hash* function, allocate a 64-bit register for each *Key*, and set the size of the register array to store results temporarily as 16.

The control plane part is written in ∼3K lines of C code. It is responsible to initialize the data plane, inject template packets, receive update notifications, update entries/registers in the data plane and interact with the campus monitoring systems. In the probe packet generation module, we set β in Equation (2) as 0.8 to accommodate to traffic bursts, and solve the LP problem with the Gurobi [18] toolboxes. Since the routing tables in our campus network are pretty stable, we only estimate the extra load $d_{p,e}$ on each link by full-rate probe packets once, with the approach in §4.1.2. In the response packet processing module, to reduce the risk of suffering from chosen plaintext attacks, the control plane generates a random *Key* every t=1 second and the data plane applies Xorshift [32] as the random number generator.

Besides, the backend agent running on the storage server is implemented with DPDK, which extracts the scanning results from the aggregated response packets and writes the results into a Redis [29] database.

6 Evaluation

In this section, we evaluate IMap via testbed experiments and real-world deployments to answer the questions below:

- What is the overall effectiveness of IMap to conduct in-network scanning (§6.2)?
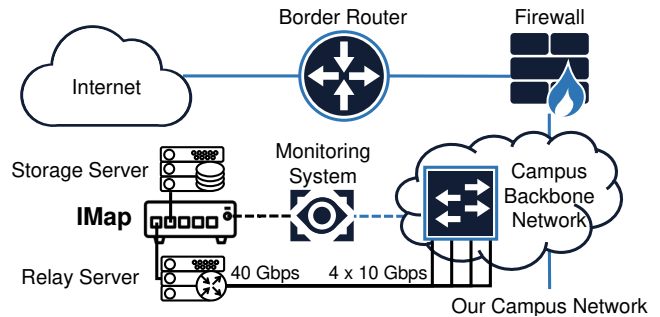
Figure 7: Deployment of IMap.

- Can IMap generate high-speed probe packets with random address and adaptive rate (§6.3)?

- Can IMap process response packets correctly and efficiently (§6.4)?

- How helpful is IMap in understanding our campus network's security situations (§6.5)?

6.1 Experimental Setup

IMap setup. Our testbed is composed of one 3.3 Tbp/s Intel Tofino switch (Edgecore Wedge 100BF-32X) and two Dell R730 servers. Both servers are equipped with Intel(R) Xeon(R) E5-2620 v3 CPUs and 64 GB memory, and connected to the switch via 40 GbE Intel XL710 NICs. In particular, one server runs as the storage server and the other server runs as the relay node to bridge IMap with our campus network. With the relay server, we can collect and analyze the probe packets from IMap and the response packets to IMap accurately. Working with the network administrator of our campus network, we deploy IMap to connect to one backbone switch in our campus network, as shown in Figure 7. Due to security and reliability considerations, we are not allowed to replace the backbone switch with the IMap switch. The network conditions are obtained from the monitoring systems in our campus network, according to which IMap adjusts its scanning rate correspondingly.

Baselines. We use two state-of-the-art network scanners as baselines in our experiments, i.e., Zipper ZMap [1] (Z-ZMap for short) and Masscan [33]. They are deployed on a Dell R430 server located at the network edge, which is equipped with a 10 GbE Intel 82599ES NIC to connect to our campus network. Note that 10 Gbps is the maximum capacity officially supported on the project homepage of these baseline scanners. We adopt the fastest configuration recommended in Z-ZMap [1] for baseline scanners to achieve the best scanning capability, e.g., we install "PF_RING ZC" NIC driver to support the high-speed scanning of Z-ZMap and Masscan.

Scanning Task. Since TCP SYN scanning is one of the most representative probes among single-packet probes, we use TCP SYN scanning to evaluate IMap in our experiments. The scanning target is configured to some or all ports (0∼65535)

Table 1: Scanning rate and scanning completion time.

Scanner	Scanning Rate	Time for 1000-Ports Scan[1]	Time for All-Ports Scan
IMap	55.6 Mpps	12 seconds	8 minutes
Z-ZMap	14.2 Mpps	35 seconds	33 minutes
Masscan	9.4 Mpps	51 seconds	50 minutes

Table 2: Switch resource utilization.

	Computational			Memory	
Resource Usage	Tables 42.86%	ALUs 45.84%	Gateways 18.75%	SRAM 20.83%	TCAM 0.69%

of our campus network including 6 Class B addresses, a total of up to 25 billion scanning space, which is nearly 6 times larger than Internet-wide single-port scanning space.

6.2 Overall Effectiveness

Scanning accuracy. In order to determine whether IMap can perform high-speed scanning and obtain accurate scanning results in our campus network, we examine whether the scanning rate, i.e., the rate of probe packets sent from IMap, has any effect on the hit rate, i.e., the fraction of responsive probed hosts (responding with SYN-ACK or RST in this case). We experiment by using IMap and baseline scanners to scan port 80 of our campus network and normalize the experimental results. Figure 8 shows that IMap is capable of handling scanning at up to 55 Mpps without obvious hit rate degradation. In contrast, baseline scanners such as Z-ZMap and Masscan can neither reach a high scanning rate, nor achieve a comparable hit rate (at least 1.5 times gap). These benefits are brought by the in-network deployment location and performant switch implementation. Our results also verify baseline scanners experience the decreased hit rate with the higher scanning rate due to the drop of probe/response packets [1].

Vantage point. To demonstrate the advantage of IMap in employing in-network scanning, we probe all addresses in our campus network on port 80 and measure the latency between sending a probe packet and receiving the response packet from active hosts. We also conduct the same measurement on two baseline scanners at the same time. The CDF of the results are shown in Figure 9. IMap gains much shorter round-trip response time for over 90 percent of hosts than state-of-the-art scanners. This is benefited from the fact that IMap is deployed in the core network and probe/response packets take a shorter path, 2-4 hops compared with 4-8 hops of end-to-end scanning. It also indicates the less bandwidth waste to the network and the smaller possibilities of dropping probe/response packets, which promises IMap can conduct high-speed scanning accurately and efficiently.

Fastness and scalability. To illustrate that IMap is fast and scalable in network scanning, we measure the scanning rate and scanning completion time of IMap and baseline scanners. Port 0-999 and all ports of our campus network are chosen as the scanning tasks respectively. For each scanner, we repeat both tasks for 10 trials at the midnight to minimize the impact for our campus network and report the averages in Table 1.

---

[1]It includes time to send probe packets as well as a fixed 5-second delay after all probes are sent, during which scanners wait for late response packets.

The results show IMap is able to generate 55.6 million probe packets per second (close to 40 Gbps linespeed), which is 4 times improvement compared with Z-ZMap and Masscan. Note that 40 Gbps is not the upper limit of IMap; instead, when we enable all ports of the switch, IMap can generate probe packets at terabit line rate. Currently, we cannot replace the core switch with IMap to conduct such a pressure test, which is left for our future work. Besides, Table 1 also shows IMap can complete scanning tasks much faster than the other scanners, which can help operators capture network security snapshots much more quickly.

Resource overhead. To evaluate the resource consumption of IMap, we focus on its resource usage of our test switch, which is a low-end switch with pretty limited resources. Table 2 displays the average hardware resource utilization of IMap across all stages of the switch. As we can see, even with such a low-end switch, IMap takes up less than half of computational resources, one-fifth of SRAM, and negligible TCAM, still leaving enough resources for the concurrent execution of traditional forwarding behaviors [35]. Leveraging high-end switches with more hardware resources (e.g., Edgecore Wedge 100BF-65X), the resource usage of IMap can be much lower. Besides, the resource utilization of a switch does not have any obvious effect on its forwarding performance. This is because as long as the compiled P4 program that integrates IMap and the forwarding functionality can be fitted into the switching ASICs, the switch is guaranteed to process and forward packets at terabit line rate [26, 51].

6.3 Probe Packet Generation

Random probing. To validate the randomness of probe addresses generated by IMap, we first explore the distribution of the first 1000 addresses selected by IMap and Z-ZMap when they are probing port 80 of our campus network. Considering our campus network only contains class B addresses, as shown in Figure 10, we only keep the last two octets of the IP address and map them to the x and y coordinates, respectively. Based on the results, we can see that the address randomization of IMap achieves slightly worse statistical properties than Z-ZMap because IMap employs the PIPR table, but we believe it is still good enough to avoid overwhelming the destination networks. To verify this, we analyze the pressure IMap brings to access networks. Figure 11 indicates several vital access networks in our campus network only receive thousands of probe packets per second even though the scanning rate of IMap reaches as high as 55 Mpps. Such additional packet overhead is negligible for most edge networks.
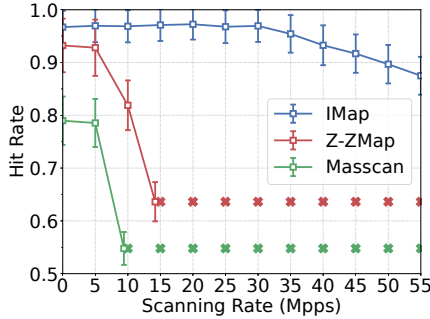
Figure 8: Hit rate vs. Scanning rate.
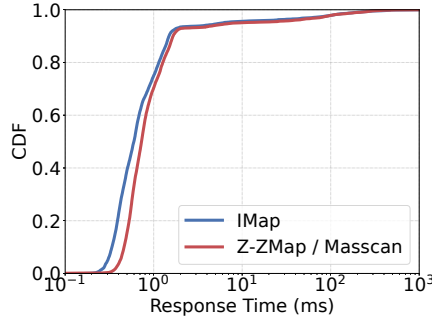
Figure 9: Response time of probe packets.
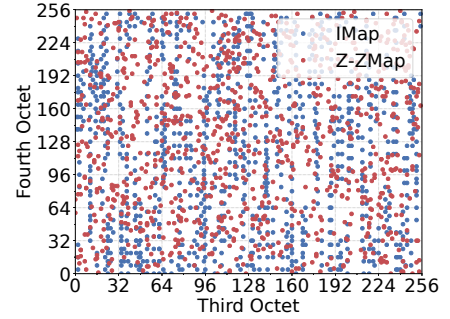
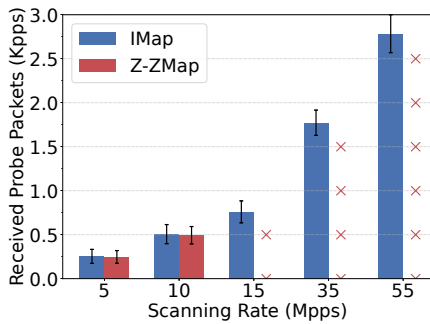Figure 10: Distribution of generated probe addresses.
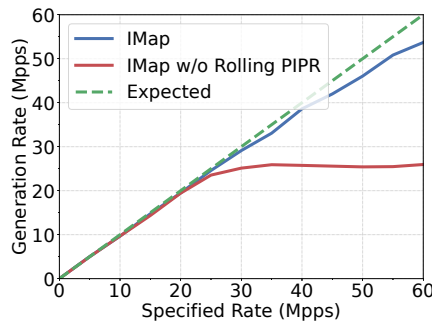

Figure 11: Network pressure for access networks.
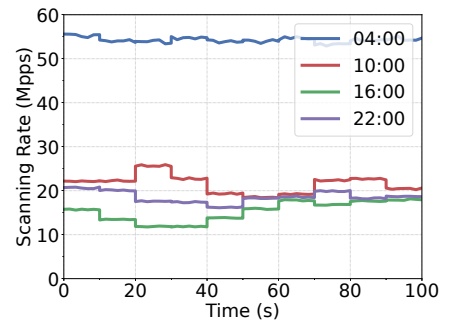

Figure 12: Generation rate of probe packets.

Figure 13: Adaptive scanning rate.

Adaptive probing. To evaluate the adaptability of scanning rate of IMap, we first quantify the rate control accuracy of IMap by comparing the rate specified by the runtime parameter with the actual rate of probe packets sent from IMap. As shown in Figure 12, the error gradually increases with the rising of the scanning rate, but it is always limited within 5% even when the scanning rate of IMap reaches 55 Mpps. Such error mainly comes from the restricted accuracy of the packet rate in the recirculate port and can be manually corrected in the real scanning. Besides, from this figure, we can also see that rolling PIPR filling optimization (§4.1) helps IMap achieve high-speed scanning continuously. Then we investigate whether IMap can adjust its scanning rate according to network conditions. We conduct scanning on our campus network with IMap at different time, and record the rate of probe packets in Figure 13. Since the monitoring system reports the campus network conditions every 10 seconds, and the LP problem can be solved within 3 seconds for our campus network, we make IMap update the scanning rate every 10 seconds to adapt to the change of the network conditions.

6.4 Response Packet Processing

Secure verifying. The security of the response verifier is guaranteed by the dynamic key updating technique in IMap, whose efficiency is decided by the parameter t. To find a suitable value for t, we first simulate the relationship between the computing power and the time required to reverse *Key*

used by the hash function in IMap. As we can see from Figure 14, it takes about 4 seconds for high-end CPUs and more than 20 seconds for mainstream CPUs to locate the real *Key* using the stack algorithm [38]. In this case, IMap is protected from chosen plaintext attacks with t smaller than 1.3 seconds. Then we choose several different t for IMap and scan all ports of our campus network to seek how t affects probing. Figure 15 presents the number of response packets received by IMap but not pass the verifier during each scan, which occurs when the response time is beyond 3t. The results manifest that, under a common attacker, 0.3s~1.3s are all applicable choices for t in our campus network.

Response aggregating. To testify the efficiency of response packet aggregation mechanism, we configure IMap to scan the campus network at different rate, and monitor the response traffic that is sent from the switch to the storage server. Figure 16 and 17 display the packet rate and throughput of such traffic with or without aggregating response packets respectively. It can be seen that the aggregation enables a 93.8% reduction in RX rate and an 86.1% reduction in RX throughput for the storage server, which efficiently protects it from being saturated by massive response traffic.

6.5 Analysis of Scanning Results

High-speed scanning of IMap has enabled the faster snapshots of the network. Therefore, we conduct an experiment where IMap continuously scans all addresses in our campus
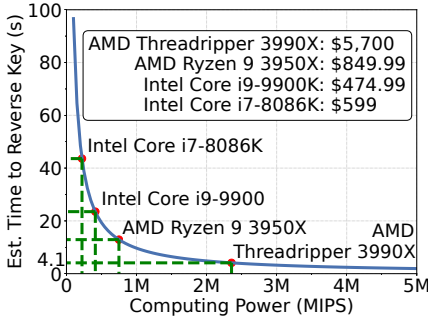
Figure 14: Reverse time for *Key*.
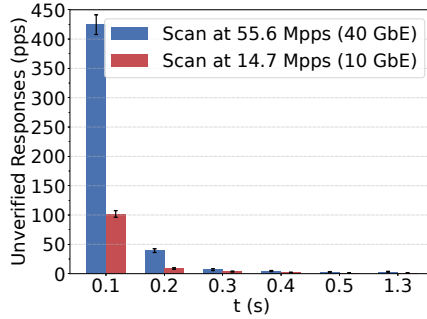
Figure 15: Impact of t to probing.
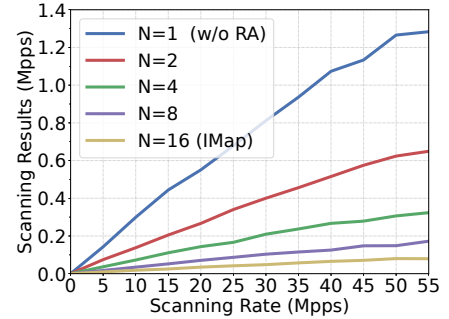
Figure 16: Packet rate of scanning results.
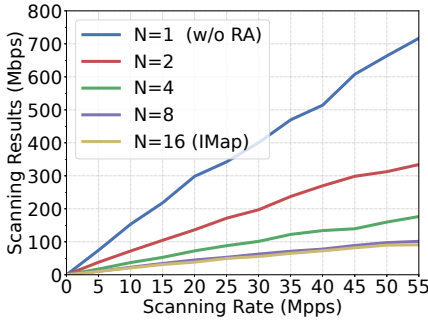
Figure 17: Throughput of scanning results.
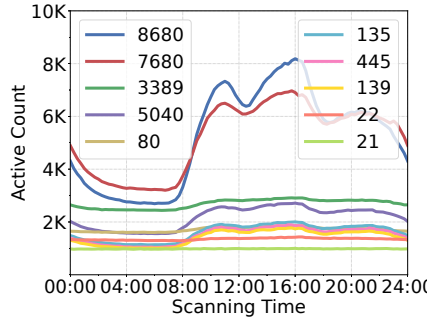
Figure 18: Activity of ports over one day.
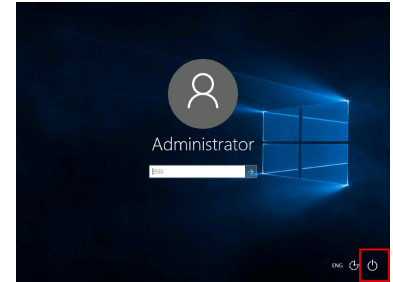
Figure 19: RDP screenshot of an vulnerable host.

Table 3: Top 10 active TCP ports of our campus network.

Port	Service	Active Rate	Active Count	Inactive Count
8680	WeChat	1.34%	5271	12555
7680	Windows Update Delivery	1.33%	5211	12065
3389	RDP (Remote Desktop)	0.69%	2693	11659
5040	Windows Deployment Service	0.55%	2176	11709
80	HTTP	0.44%	1722	4841
135	Microsoft DCE/RPC	0.41%	1607	11081
445	Microsoft-DS	0.38%	1499	10592
139	NetBIOS Session	0.36%	1422	10559
22	SSH	0.34%	1354	4831
21	FTP	0.25%	983	10918

Table 4: Exploitability of vulnerabilities to 135 and 3389.

Port	Vulnerability	Exploitability
135	Leak the host name, OS version, timestamp	100%
	Leak all NICs and IPs	99.6%
	Leak all RPC services	98.8%
3389	Leak the host name, OS version, timestamp	81.3%
	Leak the login screen	35.4%
	Remote shutdown*	20.2%

network on all TCP ports. This experiment lasts for a week and the scanning results in the Redis database are persisted into the disk with a tag of time after each scan is over. In order to explore potential applications of IMap, based on the scanning results, we attempt to track the adoption of common protocols and discover new potential risks and security incidents in our campus network.

Protocol adoption. We first compute the average count of active and inactive hosts for each port in all time periods. Table 3 lists the top 10 open ports we observed and reveals several interesting findings. First, as the proportion of online devices in our campus network ($\sim$5%) is far lower than that of the Internet, the active rate of the port is also lower than that of the Internet. Besides, we notice IMap just receives a small number of response packets from some sensitive ports, like ports 22 and 80, and we speculate the reason is that many

systems filter such probe packets via the host firewall. Finally, we find Table 3 displays a really different sorting from that of the Internet in ZMap [14]. For example, the most active port in our campus network is 8680, which is used by WeChat, one of the most popular messaging App, and the second one is 7680, which is occupied by Windows to distribute system updates. We also observe a surprising number of open ports associated with file/device sharing over the network, such as ports 139, 445, and 5070. We attribute these differences to that more personal devices than servers are connected to our campus network. Furthermore, we then analyze the active rate variation of the top 10 ports by time over one day. As we can see in Figure 18, the active rate of some ports, e.g., 8680 and 7680, exposes an obvious diurnal pattern while that of other ports does not change significantly over time. This is because the former are usually opened by personal devices while the latter are opened by servers.

Potential risks. Among the top 10 ports, 135 (DCE/RPC) and 3389 (RDP) catches our attention because they are known for information leakage. Considering their popularity, we investi-

gate the exploitability of their vulnerabilities in our campus network. As shown in Table 4, 100% of the 135-opened hosts and more than 80% of the 3389-opened hosts are at the risk of information leakage. Moreover, the 3389-opened Windows hosts are also vulnerable to being shutdown remotely due to the misconfiguration from their users. For instance, Figure 19 is the screenshot from one of such vulnerable hosts, showing that users are allowed to perform shutdown operations on the RDP login screen. Even though the firewall of our campus network bans external access to internal hosts' sensitive ports including 135 and 3389, we believe these vulnerabilities still pose a high risk to our campus network and may be exploited by attackers. We have contacted our network administrators, and they confirmed these risks and issued a notice to remind teachers and students to check their configurations.

Botnets detection. We also implement several alarm scripts triggered when the scanning results satisfy some conditions. One of them is used to detect botnets by monitoring whether the active count of certain ports surges in the last scan. During our experiment, we did find a fast increase of 48101-opened hosts and suspected it is caused by a Mirai botnet. We reported such an issue to the network administrators immediately and they finally determined it is just an experiment on Mirai conducted by a security lab. Even we dodged a bullet, it still reflects the potential of IMap in fast revealing security incidents with high-speed scanning, which cannot be obtained in time by existing network scanners like Z-ZMap and Masscan.

7 Discussion

Scanning results v.s. deployment locations. From a network perspective, different switches have diverse network utilization, topological connection relations and access restrictions. As a result, the deployment locations of IMap affect the scanning results inevitably. Furthermore, we can also coordinate multiple switches to deploy IMap for cooperative scanning, which can achieve a higher scanning rate and hit rate. For any given network, there must be optimal distributed deployment locations in a given period of time, which can achieve the highest scanning rate and hit rate. We leave the deep exploration of optimal distributed deployment locations in a given network as our future work.

Relationship with application-layer scanners. Currently, IMap only supports single-packet scanning, including TCP SYN scans, ICMP echo request scans, and application-specific UDP scans, and does not support complex application-layer protocols, e.g., TLS handshakes, directly. However, similar to ZMap, IMap can serve as a foundation to obtain the responsive host list from the given port, e.g., port 443 for TLS protocol. Based on this list, operators can use application-layer scanners to collect advanced information, e.g., a custom certificate fetcher to retrieve TLS certificates. In a word, IMap can narrow down the scanning space for application-layer scanners significantly.

8 Related Works

Our work is highly related to the following topics.
Network scanners. Many network scanners have been developed to conduct network scanning tasks. Nmap [39] is optimized for small network segments with a wide variety of probing techniques. IRLscanner [30], ZMap [14], Masscan [33] and Zipper ZMap [1] are designed for Internet-scale scanning, mainly with a single-packet probing paradigm. IMap is very similar to ZMap and Masscan in the scanning methodology, but with different implementation targets and deployment locations, thus achieving orders of magnitude scanning capability improvement.

IPv6 scanning. Numerous research works have been devoted to improving the IPv6 scanning efficiency by optimizing the scanning space algorithmically. Entropy/IP [15] employs information entropy to segment the addresses in the hitlist and generate target addresses based on the relationship between different fragments. 6Gen [36] and Entropy-Clustering [16] extend the scope of prefix space for Entropy/IP and discover seed address fingerprint with clustering analysis. 6hit [21] adopts a reinforcement learning based target generation method to improve the probing efficiency. As a high-speed scanning system, IMap is completely orthogonal to these algorithmic works. And the scanning space generated from these algorithms can be set as the input of IMap to further improve the scanning efficiency.

Programmable switches. Recently programmable switches have been used as accelerators for various applications in networking [17, 35, 37], distributed systems [25, 26] and security [27, 34, 51], and these applications achieve far better performance with lower costs than their software counterparts running on commodity servers. The closer work to ours is HyperTester [49], which shows how to design a high-speed network tester with programmable switches. However, HyperTester neither illustrates how to generate probe packets with random address and adaptive rate, nor how to process response packets correctly and efficiently. IMap addresses these unique challenges, and thus turns a switch into a practical high-speed network scanner.

9 Conclusion

In this paper, we identify the limitations of current network scanners, and introduce IMap, a fast and scalable in-network scanner with programmable switches. We devise a set of techniques and optimizations, i.e., an address-random and rate-adaptive probe packet generation mechanism, and a correct and efficient response packet processing mechanism, to turn a switch into a practical high-speed network scanner. We implement an open-source prototype of IMap and conduct extensive evaluations to show the advantages of IMap compared with current network scanners. We hope IMap can serve as the foundation of next-generation terabit network scanners.

References

[1] David Adrian, Zakir Durumeric, Gulshan Singh, and J Alex Halderman. Zippier zmap: internet-wide scanning at 10 gbps. In *8th USENIX Workshop on Offensive Technologies (WOOT 14)*, San Diego, CA, 2014. USENIX.

[2] Johanna Amann, Oliver Gasser, Quirin Scheitle, Lexi Brent, Georg Carle, and Ralph Holz. Mission accomplished? https security after diginotar. In *Proceedings of the 2017 Internet Measurement Conference*, pages 325–340, New York, USA, 2017. ACM.

[3] Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J Alex Halderman, Viktor Dukhovni, et al. Drown: Breaking tls using sslv2. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 689–706, Austin, TX, 2016. USENIX.

[4] AVSystem. 5g iot: What does 5g mean for iot? https://www.avsystem.com/blog/5g-iot/, 2021.

[5] D. J. Bernstein. Syn cookies. https://cr.yp.to/syncookies.html, 2021.

[6] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. A messy state of the union: Taming the composite state machines of tls. In *2015 IEEE Symposium on Security and Privacy*, pages 535–552, San Jose, CA, USA, 2015. IEEE, IEEE.

[7] Robert Beverly, Ramakrishnan Durairajan, David Plonka, and Justin P Rohrer. In the ip of the beholder: Strategies for active ipv6 topology discovery. In *Proceedings of the Internet Measurement Conference 2018*, pages 308–321, 2018.

[8] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[9] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. *ACM SIGCOMM Computer Communication Review*, 43(4):99–110, 2013.

[10] Stephen Checkoway, Ruben Niederhagen, Adam Everspaugh, Matthew Green, Tanja Lange, Thomas Ristenpart, Daniel J Bernstein, Jake Maskiewicz, Hovav Shacham, and Matthew Fredrikson. On the practical exploitability of dual ec in tls implementations. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 319–335, San Diego, CA, 2014. USENIX.

[11] Cisco. Best practices in core network capacity planning white paper. https://www.cisco.com/c/en/us/products/collateral/routers/wan-automation-engine/white_paper_c11-728551.html, 2021.

[12] Intel DPDK. Learn how to get involved with dpdk. https://www.dpdk.org/, 2021.

[13] Zakir Durumeric, David Adrian, Ariana Mirian, James Kasten, Elie Bursztein, Nicolas Lidzborski, Kurt Thomas, Vijay Eranti, Michael Bailey, and J Alex Halderman. Neither snow nor rain nor mitm... an empirical analysis of email delivery security. In *Proceedings of the 2015 Internet Measurement Conference*, pages 27–39, New York, USA, 2015. ACM.

[14] Zakir Durumeric, Eric Wustrow, and J Alex Halderman. Zmap: Fast internet-wide scanning and its security applications. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 605–620, Washington, D.C., USA, 2013. USENIX.

[15] Pawel Foremski, David Plonka, and Arthur Berger. Entropy/ip: Uncovering structure in ipv6 addresses. In *Proceedings of the 2016 Internet Measurement Conference*, pages 167–181, 2016.

[16] Oliver Gasser, Quirin Scheitle, Pawel Foremski, Qasim Lone, Maciej Korczyński, Stephen D Strowes, Luuk Hendriks, and Georg Carle. Clusters in the expanse: Understanding and unbiasing ipv6 hitlists. In *Proceedings of the Internet Measurement Conference 2018*, pages 364–378, 2018.

[17] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 conference of the ACM special interest group on data communication*, pages 357–371, 2018.

[18] Gurobi. The fastest mathematical programming solver. http://www.gurobi.com/, 2021.

[19] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J Alex Halderman. Mining your ps and qs: Detection of widespread weak keys in network devices. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 205–220, Bellevue, WA, 2012. USENIX.

[20] Ralph Holz, Johanna Amann, Olivier Mehani, Matthias Wachs, and Mohamed Ali Kaafar. Tls in the wild: An internet-wide analysis of tls-based protocols for electronic communication. In *Symposium on Network and Distributed System Security (NDSS)*, San Diego, CA, USA, 2016. Internet Society.

[21] Bingnan Hou, Zhiping Cai, Kui Wu, Jinshu Su, and Yinqiao Xiong. 6hit: A reinforcement learning-based approach to target generation for internet-wide ipv6 scanning. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*, pages 1–10. IEEE, 2021.

[22] IMapScanner. IMap. https://github.com/IMapScanner/IMap.git, 2021.

[23] Intel. Intel Tofino: P4-programmable Ethernet switch ASIC that delivers better performance at lower power. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html, 2021.

[24] Google Ipv6. Ipv6 adoption. https://www.google.com/intl/en/ipv6/statistics.html, 2021.

[25] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 35–49, Renton, WA, USA, 2018. USENIX.

[26] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 121–136, 2017.

[27] Qiao Kang, Lei Xue, Adam Morrison, Yuxin Tang, Ang Chen, and Xiapu Luo. Programmable in-network security for context-aware byod policies. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 595–612, 2020.

[28] Praveen Kumar, Yang Yuan, Chris Yu, Nate Foster, Robert Kleinberg, Petr Lapukhov, Chiun Lin Lim, and Robert Soulé. Semi-oblivious traffic engineering: The road not taken. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 157–170, 2018.

[29] Redis Labs. Redis. https://redis.io/, 2021.

[30] Derek Leonard and Dmitri Loguinov. Demystifying service discovery: implementing an internet-wide scanner. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 109–122, 2010.

[31] Hongqiang Harry Liu, Srikanth Kandula, Ratul Mahajan, Ming Zhang, and David Gelernter. Traffic engineering with forward fault correction. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, pages 527–538, 2014.

[32] George Marsaglia et al. Xorshift rngs. *Journal of Statistical Software*, 8(14):1–6, 2003.

[33] Masscan. Masscan: Mass ip port scanner. https://github.com/robertdavidgraham/masscan, 2021.

[34] Roland Meier, Petar Tsankov, Vincent Lenders, Laurent Vanbever, and Martin Vechev. Nethide: Secure and practical network topology obfuscation. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 693–709, 2018.

[35] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 15–28, 2017.

[36] Austin Murdock, Frank Li, Paul Bramsen, Zakir Durumeric, and Vern Paxson. Target generation for internet-wide ipv6 scanning. In *Proceedings of the 2017 Internet Measurement Conference*, pages 242–253, 2017.

[37] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 85–98, 2017.

[38] Gabriel Nivasch. Cycle detection using a stack. *Information Processing Letters*, 90(3):135–140, 2004.

[39] NMAP.ORG. Nmap. https://nmap.org/, 2021.

[40] Tian Pan, Nianbing Yu, Chenhao Jia, Jianwen Pi, Liang Xu, Yisong Qiao, Zhiguo Li, Kun Liu, Jie Lu, Jianyuan Lu, et al. Sailfish: accelerating cloud-scale multi-tenant multi-service gateways with programmable switches. In

Proceedings of the 2021 ACM SIGCOMM 2021 Conference, pages 194–206, 2021.

[41] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. Netbricks: Taking the v out of nfv. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 203–216, 2016.

[42] Philipp Richter, Georgios Smaragdakis, David Plonka, and Arthur Berger. Beyond counting: new perspectives on the active ipv4 address space. In *Proceedings of the 2016 Internet Measurement Conference*, pages 135–149, New York, USA, 2016. ACM.

[43] SDXcentral. At&t picks barefoot networks for programmable switches. https://www.sdxcentral.com/articles/news/att-picks-barefoot-networks-programmable-switches/2017/04/, 2021.

[44] SDXcentral. Barefoot scores tofino deals with alibaba, baidu, and tencent. https://www.sdxcentral.com/articles/news/barefoot-scores-tofino-deals-with-alibaba-baidu-and-tencent/2017/05/, 2021.

[45] The ZMap Team. The zmap project. https://zmap.io/, 2021.

[46] Vybint. 15 alarming cyber security facts and stats. https://www.cybintsolutions.com/cyber-security-facts-stats/, 2021.

[47] W3Techs. Usage statistics of ipv6 for websites. https://w3techs.com/technologies/details/ce-ipv6, 2021.

[48] Sophia Yoo and Xiaoqi Chen. Secure keyed hashing on programmable switches. In *Proceedings of the ACM SIGCOMM 2021 Workshop on Secure Programmable network INfrastructure*, pages 16–22, 2021.

[49] Dai Zhang, Yu Zhou, Zhaowei Xi, Yangyang Wang, Mingwei Xu, and Jianping Wu. Hypertester: high-performance network testing driven by programmable switches. *IEEE/ACM Transactions on Networking*, 2021.

[50] Menghao Zhang, Jun Bi, Kai Gao, Yi Qiao, Guanyu Li, Xiao Kong, Zhaogeng Li, and Hongxin Hu. Tripod: Towards a scalable, efficient and resilient cloud gateway. *IEEE Journal on Selected Areas in Communications*, 37(3):570–585, 2019.

[51] Menghao Zhang, Guanyu Li, Shicheng Wang, Chang Liu, Ang Chen, Hongxin Hu, Guofei Gu, Qi Li, Mingwei Xu, and Jianping Wu. Poseidon: Mitigating volumetric ddos attacks with programmable switches. In *the 27th Network and Distributed System Security Symposium (NDSS 2020)*, 2020.

Unlocking the Power of Inline Floating-Point Operations on Programmable Switches

Yifan Yuan
UIUC

Omar Alama
KAUST

Jiawei Fei
KAUST & NUDT

Jacob Nelson
Microsoft Research

Dan R. K. Ports
Microsoft Research

Amedeo Sapio
Intel

Marco Canini
KAUST

Nam Sung Kim
UIUC

Abstract

The advent of switches with programmable dataplanes has enabled the rapid development of new network functionality, as well as providing a platform for acceleration of a broad range of application-level functionality. However, existing switch hardware was not designed with application acceleration in mind, and thus applications requiring operations or datatypes not used in traditional network protocols must resort to expensive workarounds. Applications involving floating point data, including distributed training for machine learning and distributed query processing, are key examples.

In this paper, we propose FPISA, a floating point representation designed to work efficiently in programmable switches. We first implement FPISA on an Intel Tofino switch, but find that it has limitations that impact throughput and accuracy. We then propose hardware changes to address these limitations based on the open-source Banzai switch architecture, and synthesize them in a 15-nm standard-cell library to demonstrate their feasibility. Finally, we use FPISA to implement accelerators for training for machine learning as an example application, and evaluate its performance on a switch implementing our changes using emulation. We find that FPISA allows distributed training to use one to three fewer CPU cores and provide up to 85.9% better throughput than SwitchML in a CPU-constrained environment.

1 Introduction

The rise of programmable network devices has transformed distributed systems design. Instead of simply moving data between servers using standard routing protocols, network devices can be programmed using domain-specific languages like P4 [8] and NPL [10] to support new network functionality, such as congestion control [100], load balancing [55, 78], and packet scheduling [103]. Commodity Ethernet switch ASICs with programmable data planes [11, 42, 92] enable the execution of these programs at many terabits per second.

While these capabilities were originally targeted at increasing network functionality, much recent work has explored their utility in accelerating application-level functionality as well. Consensus protocols [17, 65, 93], concurrency control [45, 64], vector addition [75, 97, 98], query processing operators [34, 63], and key-value stores [49, 66, 112] have all

been shown to benefit from this in-network computation [94].

However, an important class of applications has struggled to take advantage of in-network computation: those using floating point (FP) values. These occur in two broadly-deployed datacenter applications: distributed training for machine learning, and distributed data processing systems. Since programmable switches were originally optimized for networking applications, their design includes basic support only for integer operations. Applications wanting to take advantage of in-network computation with floating point values have so far worked around this in one of three ways.

The first approach is to approximate floating point operations in software running on end-hosts. This is the approach taken by SwitchML [98] as it sums gradient vectors as part of training deep neural networks. For each chunk of gradient vector elements, SwitchML executes a protocol that requires running code to convert between floating point and integer values on end hosts, as well as performing two rounds of communication. This protocol overhead is costly (see Sec. 5.3.3).

The second approach is to build a switch ASIC that includes floating point hardware. This is the approach taken by the Mellanox Quantum switch [32, 76]. Dedicating chip resources for this purpose is expensive: we show (Sec. 4.2) that adding dedicated FPU hardware takes more than $5\times$ the die area and power of integer ALUs. As a result, this is not a general-purpose approach; it has only been taken for InfiniBand switches, which have simpler routing designs and buffer requirements than Ethernet switches, and hence have spare die area. It also lacks flexibility: it is tied to specific operations on specific floating-point formats. New ML-specific numeric representations (*e.g.*, FP16 [79, 106], bfloat16 [21, 30, 53], TF32 [86], and MSFP [18]) represent an area of ongoing innovation, and adding support for a new format requires developing and manufacturing a new ASIC – an expensive and time-consuming endeavor. For example, it took four years for Mellanox to release its second version of switches with floating point support [31, 32].

A related approach is to use FPGAs or other non-switch programmable devices to implement switch-like specialized accelerators [5, 20, 27, 70]. While this yields a functional solution, the fine-grained programmability of a FPGA comes at the cost of power [111] and area: for example, Xilinx's flagship FPGA

supports ∼8 Tbps [114] of Ethernet I/O, while the Intel Tofino 2, a regular programmable switch, supports 12.8 Tbps [43].

In this paper, we argue for a different approach. We propose FPISA, which implements floating point computation as a P4 program running directly on a programmable switch. This is not straightforward: the multi-cycle nature of floating-point operations is at odds with the streaming-pipeline architecture common to P4-programmable switches today. To make it work, FPISA breaks apart each floating point value into *exponent* and *signed mantissa* and stores them separately in different pipeline stages, decomposing the corresponding sub-operations appropriately to ensure correct execution. Rather than requiring specialized floating-point hardware, FPISA repurposes network-oriented hardware elements in the switch pipeline to implement the sub-operations not supported by the switch's integer ALUs.

FPISA is a generic approach. We evaluate its feasibility on the Intel Tofino [42], a commercially-available PISA switch. We observe that constraints of the existing Tofino architecture present obstacles to a full FPISA implementation. We address this in two ways. First, we introduce an approximate FPISA design (FPISA-A) that is implementable on existing hardware, albeit with some precision and throughput limitations. Second, we propose some simple and cheap hardware modifications, based on the open-source Banzai [102] switch architecture, to enable high throughput and accuracy with FPISA. We show that such enhancements are feasible in a 15-nm standard-cell library with minimal power, area, and timing cost relative to a baseline switch chip.

Through an emulation-based study, we assess the performance benefits of our approach by implementing accelerators for the use case of distributed training for machine learning, based on the recent SwitchML [98] framework. Enhancing SwitchML with FPISA (based on both regular FP32 and ML-specific FP16) allows it to use 1-3 fewer CPU cores, giving up to an 85.9% improvement in training throughput on CPU-limited configurations, while still achieving the same training accuracy and convergence.

2 Background and Challenges

Conventional network switches are fixed-function, requiring redesign to add new features or support new protocols. However, in today's era of software-defined networking [58], rapidly evolving networking techniques and applications require new packet processing support. Programmable switches, which allow the data plane behavior to be reconfigured, provide the necessary flexibility. The RMT-based Protocol-Independent Switch Architecture (PISA) [9] has emerged as the de facto standard for programmable switch architecture.

2.1 PISA

We depict the basic protocol-independent switch architecture design in Fig. 1. The parser is a programmable state machine responsible for extracting user-specified fields of the inbound
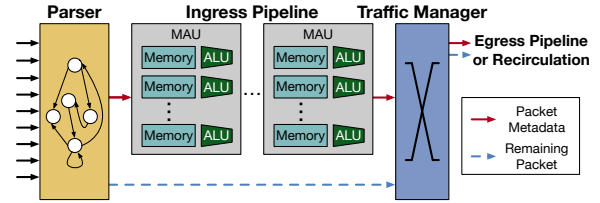
Figure 1: Basic PISA design.

packet to per-packet metadata.[1] The ingress pipeline consists of multiple cascaded match-action units (MAUs). Each MAU has some memory (SRAM and TCAM) and ALUs. It matches fields from the packet metadata against the memory to determine the corresponding action to be taken by the ALUs. The ALUs support basic integer arithmetic and logic operations, and can be used to modify fields in the packet metadata. They can also manipulate registers, which hold state that persists across different packets.

After going through the ingress pipeline, the packet is routed to an egress port and queued by the traffic manager. Before being output, it passes through an egress pipeline that has the same structure as the ingress pipeline, and the packet header and body are reassembled by the deparser.

Programmable switches following this architecture have become commercially available on commodity switches, thanks to reconfigurable switch silicon like the Intel (Barefoot) Tofino [42] and Marvell XPliant [88]. A long line of research has showed how to use PISA switches to implement new networking protocols, offload network functions, and accelerate application-level logic [36, 94].

2.2 Floating Point Overview

We describe the flow of the most common floating point operation in applications discussed in this paper – addition – here. Note that subtraction is performed using the same process, and comparisons are typically implemented using subtraction. Regardless of specific widths, floating point values are represented with three parts: 1-bit sign, n-bit exponent, and m-bit mantissa. Typically, a floating point number is represented in normalized form: the mantissa value is in the range of $[1,2)$, *i.e.*, it begins with a leading "1" bit (which can be omitted, *i.e.*, "implied 1"). A floating point addition $C = A + B$ is performed using a five-step process: (We assume here that abs(A) ≤ abs(B).)

Extract. The three parts of A and B are extracted from the packed data. The implied "1" in the packed mantissa is expressed explicitly.

Align. The two mantissas are aligned to represent values at the same magnitude. Specifically, $mantissa_A$ (the smaller one) is right-shifted by $exponent_A - exponent_B$ bits.

Add/subtract. Now that the two mantissas are aligned, they are added or subtracted, depending on sign: $mantissa_C = mantissa_B \pm mantissa_A$.

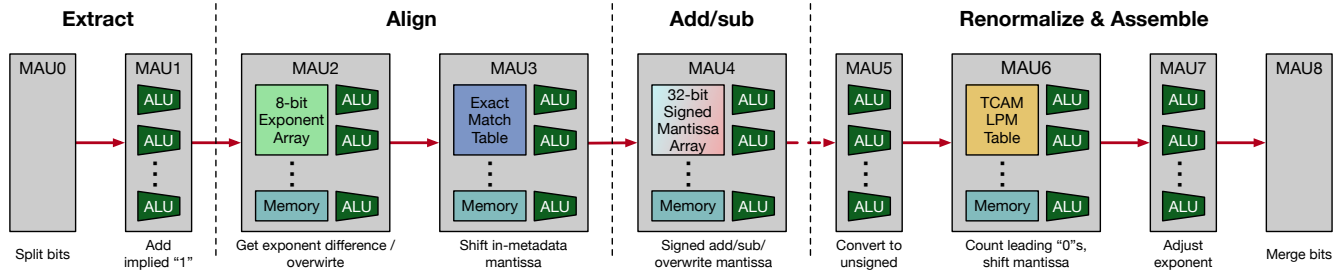[1]The remainder of the packet is passed through the pipeline, but cannot be matched or manipulated.

Figure 2: FPISA dataflow. Only hardware components relevant to FPISA are shown.

Renormalize. The result is scaled so that the mantissa is in the range of $[1,2)$. This is achieved by counting the leading "0" bits and left or right shifting $mantissa_C$ accordingly, then adjusting $exponent_C$ by the corresponding value.

Round and Assemble. Finally, the three parts of C are packed into a single value. The implied leading "1" of $mantissa_C$ is stripped. If more mantissa bits are available than can be represented in the packed format, the mantissa is rounded.

2.3 Challenges

Current PISA architectures do not natively support any floating point operations. This is no surprise, considering that they were designed for packet processing, and floating point support is expensive. FPUs have much larger power and area costs than integer ALUs [62, 68, 74], and the complex floating point addition procedure (Sec. 2.2) takes multiple cycles and thus introduces timing constraints.

This paper asks if we can build floating point addition operations on a commodity PISA architecture. Intuitively, it should be possible to decompose the canonical addition procedure and span it across multiple pipeline stages. However, we observe that this leads to two challenges.

First, registers are associated with specific pipeline stages, and can only be accessed from that stage. That is, each register can only be accessed once per packet, and data dependencies cannot "go backwards" to an earlier stage.[2] This poses a problem for applications, like in-network aggregation, that wish to maintain and update floating point state: it is not possible, for example, to perform the add-mantissa and renormalize steps in different pipeline stages.

Second, the available ALU operations may not be sufficient to implement all the operations necessary to implement floating point addition. For instance, on a CPU, the renormalization step might use a count-leading-zeros instruction (*e.g.*, lzcnt on x86), but we know of no PISA switch with such an instruction.

Hence, we must develop a PISA-friendly, decentralized (multi-stage) approach for floating point addition.

3 FPISA Design

How can we implement floating point operations on PISA architectures, given the challenges described above? We propose a design, FPISA, based on a new floating point

representation and a mapping of its operations to PISA pipelines, as shown in Fig. 2. In this section, we describe the basic FPISA approach in the context of an abstract PISA pipeline; Sec. 4 discusses additional challenges that occur when implementing it on existing PISA architectures.

FPISA has three key ideas:

Decoupled exponent and mantissa operations. FPISA processes operations on the exponent and (signed) mantissa components of floating point values separately, and internally stores them in separate registers. This decoupling allows them to be processed by different pipeline stages.

Delayed renormalization. Second, FPISA does not require intermediate values to be renormalized on every operation. That is, in a SwitchML-like [98] aggregation workflow, values from each client are added to an accumulator whose value is not renormalized until the final result is output. This is based on two observations about floating point renormalization. First, renormalization does not affect the correctness of floating point operations. Scaling the mantissa to place the leading "1" in its correct location is needed to produce an output value in canonical format, but a denormalized form can equally represent the same arithmetic value. Second, renormalization introduces data dependencies between the mantissa and exponent components, which makes it challenging to fit into a PISA pipeline. In particular, renormalization requires the exponent to be adjusted based on the computed mantissa, whose computation itself depends on the exponent – a circular data dependency that cannot be represented in a single pipeline traversal. To avoid this, when we read from the accumulator, we read the denormalized value, and normalize it just before sending out the final result. We do not store the normalized value back into the accumulator.

Extra bits in mantissa register. PISA architectures commonly have registers with limited bit widths: 8-, 16-, or 32-bit registers are common; on the other hand, floating point values commonly have mantissas with smaller bitwidth. We take advantage of this difference in two ways. First, we can use bits to the right of the mantissa as guard bits to aid in rounding, as is common in standard FPUs. Second, we can use bits to the left of the mantissa to avoid overflow when summing multiple values with similar exponents. When we add two values with mantissas that are all ones, the addition simply carries into the bits to the left of the mantissa.

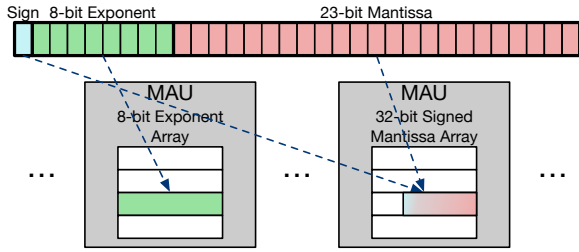In this section, we use IEEE 754 FP32 – which has a 1-bit

---

[2]Recirculating an entire packet is an exception. However, it is costly and bandwidth constrained.

Figure 3: FPISA's representation of FP32 in the switch.

Figure 4: Example of FPISA addition: computing the sum of 3.0 ($0b1.1 \times 2^1$) and 1.0 ($0b1 \times 2^0$). Computation is done using a 32-bit mantissa; 21 trailing zero bits are elided.

sign, 23-bit mantissa, and 8-bit exponent – as an example to demonstrate FPISA design. Other FP formats with different widths can also be supported. Fig. 2 shows FPISA's dataflow.

3.1 Representing FP in PISA

To meet the constraints of PISA, FPISA splits the storage of floating point values using the representation shown in Fig. 3. The exponent field is stored in an 8-bit-wide register array. The 23-bit mantissa is stored, right-aligned, in a 32-bit register. To unify signs and addition/subtraction operations, we store the mantissa in two's-complement signed representation.

FPISA needs more memory space to store a floating point number (*e.g.*, 8+32=40 bits for a FP32 number). However, we argue that this will not significantly reduce the efficiency of FPISA since exponent and mantissa have to be stored in different MAUs anyway. Hence, the per-MAU parallelism of floating point operations will not be affected.

3.2 Performing FP operations in PISA

By delaying renormalization until the output phase and storing exponents and mantissas separately, FPISA makes it possible to adapt the standard extract-align-add-renormalize-assemble floating point addition flow to a PISA pipeline. Fig. 2 shows the mapping of functionality to MAUs. We use a running example (Fig. 4) where an input of 1.0 is added to a register containing the value 3.0.

Extract. The first stages extract the exponent and mantissa

Match (Man$_{metadata}$)	Action (Man$_{metadata}$)
64.0.0.0/2	Right-shift 7 bits
...	...
1.0.0.0/8	Right-shift 1 bit
0.128.0.0/9	Do nothing
0.64.0.0/10	Left-shift 1 bit
...	...
0.0.0.1/32	Left-shift 23 bits
Default	Do nothing

Figure 5: LPM match-action table (MAU6) in FPISA design.

from a FP32 value in the input packet into separate metadata registers (MAU0), then add the implied "1" to the extracted mantissa field (MAU1). The decoded values are shown in Fig. 4 step (1).

Align. FPISA then compares the provided exponent value with the one stored in memory in MAU2. This updates the exponent and determines which of the two operands's mantissa must be right-shifted and by how much. The right shift itself is performed for the metadata value by MAU3, and for the memory value by MAU4 (where the mantissa register is located). In Fig. 4 step (2), 1.0 is shifted right to be expressed as 0.1×2^1

Add. In addition to shifting the mantissa of the in-memory value, MAU4 performs the mantissa addition itself. Depending on the sign bit, it either adds or subtracts the shifted mantissa value generated in the previous stage from the stored mantissa value (step (3) in Fig. 4). The resulting mantissa value replaces the previous stored mantissa.

Note that MAU4 is used both to perform the right shift of the stored mantissa and its addition. This is a necessity because the PISA architecture can only update a given register from one stage. Existing implementations may not be able to perform both operations with a single stateful ALU; we discuss how to extend them or how to work around this limitation in Sec. 4.

At the end of this process, the exponent and mantissa registers contain the result of the addition, but may not be in normalized form. For example, in step (4) of Fig. 4, the registers store the value $0b10.0 \times 2^1$. This is indeed a valid representation of the result 4.0, but is not in normalized form because the mantissa has more than one digit to the left of the binary point.

Renormalize and Assemble. FPISA delays renormalization: it does not renormalize the intermediate value stored in registers, but only when the result is to be output. Thus, multiple additions can be performed before renormalization. This offers two benefits. As mentioned before, it eliminates the need to adjust the exponent stored in memory after calculating the mantissa, avoiding a data dependency. Second, since the renormalization and assembly steps are stateless, we can place them in the (normally underutilized) egress pipeline, making more efficient use of resources.

The renormalization process itself is performed in four steps. The aggregated mantissa is first converted from its two's complement signed representation to unsigned value and sign (MAU5). FPISA then counts the number of leading zeros and

shifts the mantissa value accordingly, in order to place the leading "1" bit in the right location (MAU6).

Because no PISA switches support a count-leading-zeros operation, FPISA exploits a TCAM-based longest prefix match (LPM) table – commonly used in IP routing – to implement this function. Specifically, we construct a LPM table where each entry has an IP address with only the ith bit set, and a netmask that matches the first i bits. A match indicates that the mantissa has $i-1$ leading zeros. This is used to select the right shift action that places the leading 1 in its canonical location (bit 24 for FP32). In the example, the leading "1" is located using a match, whose bitwise representation is shown in step (5), which corresponds to the CIDR address 0.128.0.0/9; the lookup table (Fig. 5) indicates that the mantissa should be shifted right by 1. The exponent is adjusted also according to the leading zeros' count (in MAU7) – here, incremented by 1. This gives a normalized result; all that remains is to merge the sign, exponent, and lower 23 bit of the 32-bit mantissa fields (in MAU8) to put it in FP32 format.

3.3 Additional Floating Point Features and Operations

Overflow. The denormalized representation has the potential to overflow if similar values are added many times. With a signed register size of 32 bits and a mantissa size of 24 bits, there are 7 bits to the left of the mantissa available for holding overflows. This is sufficient to represent 128 additions of values with the maximum mantissa with the same exponent – an extreme case – into a single register without overflow. However, for the use cases described later in the paper, the number of operations per register is equivalent to the number of nodes in the distributed system. If overflow occurs, it can be detected and signaled to the user, who can handle it in an application-specific way.

Other FP formats. FPISA can be trivially modified to support floating point formats with different exponent and mantissa width (*e.g.* FP16, which we evaluate in Sec. 5). Likewise, block floating point formats, where multiple values share one exponent [18], can be supported by replicating the exponent register.

Rounding. For simplicity, we have described FPISA without guard digits. The combination of no guard digits and two's-complement representation provide round-toward-negative-infinity semantics. An implementation with n guard digits would simply store the mantissa shifted left n bits from what is show in Fig. 3, and would use those to perform other types of rounding after renormalization.

Reproducibility. FPISA provides reproducibility in that the same sequence of operations and values will always produce the same result. However, since FPISA performs operations in a different order than that specified in the IEEE 754 standard, the same sequence of operations and values performed on an IEEE-754-compliant CPU may yield a different result than FPISA. For the use cases we describe in this paper, IEEE 754 compliance is not a requirement.

In this paper, we have covered the two commonly-used floating point operations – addition and comparison. They are sufficient for many distributed applications. However, other more complex and costly floating point operations may be needed in the future with emerging applications (*e.g.*, congestion control [26, 54] and network security [34]). To pave the way for future PISA implementations, we briefly discuss the possibility of supporting them.

Multiplication and division. The flow of floating point multiplication is similar to that of addition in Sec. 2.2. The two major differences are (1) the two exponents are added, and (2) the two mantissas are multiplied, all as integers. For small floating point types, the mantissa multiplication can be implemented as a table lookup, without hardware modifications. For larger floating point types, integer multiplers could be added to the hardware. We implement one based on Banzai and find its overhead is acceptable: approximately the same as an adder and a boolean module w.r.t. power and area.

Floating point division has a different flow and takes more clock cycles than other basic operations [104], which means it is unsuitable to have a direct hardware implementation in programmable switches. For some use cases, division can be implemented by converting the dividend to its reciprocal at the end-host and then multiplying in the switch.

Logarithms. The core operation of a floating point logarithm is the integer logarithm of the mantissa. As prior research [3, 99, 109] shows, this can be done by a lookup table of fewer than 2000 entries with low error (<1%).

Square roots. Square roots are even more expensive and time-consuming (*e.g.*, more than 20 clock cycles) than division [69, 87, 104]. As with logarithms, we suggest a lookup-table-based approximation for this algorithm.

4 Realizing FPISA on PISA Architectures

The previous section shows how FPISA can map floating point operations to an abstract PISA architecture. Actual PISA implementations may have restrictions on MAU operations. We have implemented FPISA in P4 for the Tofino architecture. In doing so, we encountered several architectural limitations (Sec. 4.1). We show that simple architectural extensions, which can be implemented with minimal power and chip area cost, can resolve these limitations and enable a full FPISA implementation (Sec. 4.2). Alternatively, we describe an approximate approach, FPISA-A, which works around these limitations to implement a variant of FPISA for the existing Tofino architecture, albeit with tradeoffs in accuracy and resource utilization (Sec. 4.3).

4.1 Challenges

We implement FPISA addition in the P4 language [8] (~580 LoC) in a modularized manner (*i.e.*, one floating point addition per module) and compile it to the Tofino ASIC [42]. Tab. 1 shows the resource utilization of the FPISA module out of a single Tofino pipeline. Most of these resources cannot be

Table 1: FPISA resource utilization. Nine pipeline stages (out of 12 in total) are used.

Resource	Total usage	Max usage in a MAU
SRAM	1.15%	5.00%
TCAM	0.03%	4.17%
Stateful ALU	8.33%	50.00%
VLIW instruction slots	19.01%	96.88%
Input crossbar	0.09%	4.38%
Result bus	2.34%	12.50%
Hash bit	1.06%	7.93%

shared across multiple FPISA instances.

Using this implementation, we identify three limitations of the the current Tofino hardware that impact the functionality and efficiency of our FP operations.

Resource utilization of shift operations. In general, multiple FPISA modules can be deployed in parallel, sharing the same pipeline stages and overlapping with each other. For many applications, performing as many operations per packet as possible is essential to achieve high performance [98]. Unfortunately, the current Tofino architecture can only accommodate one FPISA module in its ingress pipeline, *i.e.*, only one floating point addition can be performed per packet.

After analyzing the resource utilization, we observe that the main source of overhead is performing shift operations. Specifically, FPISA needs to shift fields by a variable number of bits, in order to implement the alignment and renormalization stages. However, the Tofino ALUs can only perform shift operations with a fixed shift distance, specified as an immediate. While it is possible to emulate a variable-length shift operation with the current functionality, doing so is resource intensive. In particular, per-stage VLIW instruction utilization prevents multiple FPISA instances from sharing pipeline stages.

Lack of atomic shift-and-add. One of the pipeline stages in the abstract design (MAU4 in Fig. 2) must perform two operations: right-shifting the stored mantissa to align it with the value being added, and performing the mantissa addition. Both are stateful operations on the mantissa register, so they must be performed by the same stage's ALU. However, the Tofino's ALUs cannot perform *both* a shift and an add operation. In Sec. 4.3, we show how to work around this limitation by left-shifting the other mantissa value (from the packet metadata) instead; this allows the FPISA design to be implemented on the existing Tofino architecture, but can lead to numerical error for some workloads.

Endianness conversion. While hardly unique to FPISA, endianness conversion is a non-trivial source of overhead for FPISA applications. Network devices interpret values in network byte order (big-endian), whereas most general-purpose CPUs are little-endian. To identify and process the data correctly in the switch, endianness conversion is necessary. Traditional networking applications only need to convert byte order for headers, which are relatively small. For data-intensive in-switch applications, byte order conversion for the full payload can have high overhead. While the Tofino has functional units that can
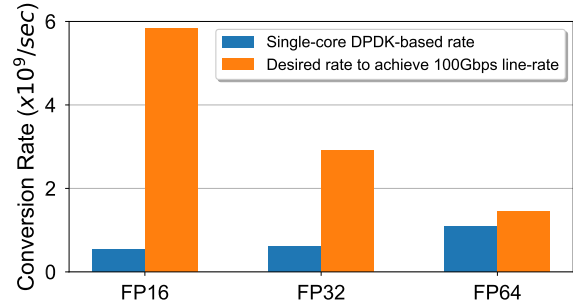
Figure 6: Endianness conversion rate that a core can achieve and that is desired to achieve 100 Gbps line-rate.

Table 2: Stateless ALU and stateful RAW/RSAW unit areas and minimum critical-path delays in FreePDK15 library. Each of the compiler targets contains 300 instances of one of the ALUs. Power and area are evaluated at 1 GHz frequency target.

	Default ALU	FPISA ALU	Default RAW	FPISA RSAW	ALU+ FPU
Dynamic power (μW)	594.2	669.4	637.6	721.1	3590.6
Leakage power (μW)	18.6	22.8	16.8	22.1	109.8
Area (μm^2)	505.4	618.6	468.8	633.0	3837.7
Min Delay (ps)	133	135	133	151	136

do this conversion, they are not plentiful enough to convert full payloads, and thus the conversion must be done on end hosts.

To quantify the overhead, we test how rapidly a single x86 core (running at 2.3 GHz) can perform endianness conversion for different floating point formats, using DPDK's highly-optimized APIs with "O3" optimization. Fig. 6 compares the measured results with the rate needed to achieve line-rate conversion at 100 Gbps. The gap is large, particularly for lower-precision values. In particular, to reach 100 Gbps for FP16, one will need at least 11 (*i.e.*, $\lceil$desired rate/single-core rate$\rceil$) cores. Hence, the high overhead of endianness conversion will lead to either low network throughput or extra CPU or GPU utilization. In many applications, these resources are not free; for instance, in DNN training, CPUs are often busy with data preprocessing.

4.2 PISA Architectural Extensions

To avoid these problems, we propose to extend the PISA architecture with some additional support. We show that the cost of these additions is low by extending the Banzai switch architecture model [102] and demonstrating that the increase in chip area, power, and timing budget is not significant.

2-operand shift instruction. We propose to enhance the existing shifter by allowing the shift distance operand to come from metadata instead of an immediate. The proposed instruction format is `shl/shr reg.distance, reg.value`. This little-effort enhancement will significantly improve the resource efficiency of FPISA, since the shifter can directly take the table match result as operand, and two instructions (left- and right-shift) can handle all the cases.

Combined shift+add operation in one stage. If the switch can support an atomic "shift+add" operation on a register in

a single stage, we will be able to swap the mantissa, with no compromise of potential error.

In-parser hardware-based endianness conversion. Endianness conversion in the hardware is straightforward and cheap – pure combinational logic shuffling the wires. We propose a simple enhancement to the switch's parser and deparser to implement this. Specifically, we propose a P4 type annotation `@convert_endianness`, applied to entire headers, that indicates to the compiler that the parser and deparser should convert the header fields' endianness as they enter and leave the pipeline. The parser will store the corresponding result to the metadata along with a implicit tag bit adjacent to the header's valid bit. When the packet is re-assembled, the deparser will check this tag bit to determine the byte order to be emitted.

To evaluate the cost of the first two changes (the last change has near-zero cost), we modify the open-source Banzai [102] switch architecture, a PISA-like design. We modify the Verilog code for Banzai's ALU to support our proposed shift instruction and synthesize it using Synopsys Design Compiler [107] with the FreePDK 15nm FinFET standard-cell library [73], a technology node similar to that used by the Tofino. We first check whether the design can operate at 1 GHz, evaluate its power and area, and then search the minimum critical-path delay of each design to find the impact of our modifcation on timing. As the results in Tab. 2 show, an enhanced ALU may use 13.0% more power and 22.4% more area than the original ALU, while slightly increasing the minimum delay. The overhead mainly comes from connecting and storing the second operand in the shifter. We implement a stateful read-shift-add-write (RSAW) unit based on Banzai's atomic predicated read-add-write (RAW) unit. The synthesis results in Tab. 2 demonstrate that the RSAW unit uses 13.6% more power and 35.0% more area than the regular RAW unit. In terms of minimum delay, RASW is 13.5% longer than RAW, but still far from the 1ns bound at 1 GHz. Banzai provides implementations only for the functional units, not for the entire switch chip, so we are unable to directly evaluate the impact of our modifications on the full chip design. However, prior work suggests that ALUs take up only a small portion (*i.e.*, $\sim 10\%$) of the power/area budget for the entire chip [9]; from this we infer that our modifications would have negligible impact. In other words, this hardware enhancement is feasible today, and is unlikely to become a bottleneck in future hardware generations.

Finally, to compare our approach with one that includes specialized floating-point units (like the Mellanox Quantum switch [32, 76]), we synthesize an ALU that includes a hard floating point unit. The ALU+FPU column in Tab. 2 shows the result: the hard FPU is more than five times larger and more power hungry than either the default ALU or the FPISA ALU. Its high area and leakage power are costs that must be paid even when the FPU is not in use, making it challenging for a switch chip including these features to be competitive with ordinary switches in terms of efficiency, and forcing vendors to maintain separate specialized switch designs for different applications.

Conversely, the FPISA approach allows the same ALUs to support both floating-point and non-floating-point computations, enabling a single switch chip design to support both floating-point and non-floating-point workloads efficiently.

4.3 FPISA-A: FPISA on Existing Architectures

The architectural changes described above allow us to implement the full FPISA approach. We additionally want a solution that allows FPISA to run on existing Tofino switches. Achieving this requires addressing the shift-and-add limitation. (The other two, while important, impact only resource utilization.) We provide a way to approximate FPISA on existing switches by avoiding the problematic shift. This approximation, which we call FPISA-A, can lead to inaccuracies for certain patterns of inputs, though we show later that it is not a problem for some applications, including in-network aggregation for ML training workloads (Sec. 5).

Recall that the problem arises because the alignment phase may require shifting the in-memory mantissa value to align it with the value to be added, which conflicts with the need to perform addition on the same value. Note that this is not a problem when the in-memory value has a larger exponent than the in-metadata value, as only the smaller of the two is right shifted. Taking advantage of FPISA's tolerance for denormalized representations, FPISA-A *always shifts the in-metadata mantissa* rather than the in-memory value. That is, if the in-metadata value is larger than the in-memory value, we keep the exponent unchanged and left-shift the in-metadata mantissa.

This approach works, within a certain range, because FPISA internally uses wider registers for the mantissa than the basic floating point representation. For FP32, IEEE 754 uses a 23-bit mantissa, while FPISA stores it in a 32-bit register. This gives 7 bits of *headroom*, after accounting for the implicit 1-bit and the sign bit. If the value being added is *much* larger than the in-memory value, i.e., its magnitude is greater by a ratio of more than $2^7 = 128$, the headroom would be exceeded. Instead, we detect this case during the exponent comparison (MAU2 in Fig. 2) and replace the in-memory value entirely with the in-metadata one. Doing so introduces numeric error in the low-order bits.

The FPISA-A variant is supported by the current commodity Tofino switch. As described above, it can introduce numeric error (which we call "overwrite" error). However, the error only occurs when input values vary widely in magnitude, and is bounded by the difference between headroom and mantissa width. For some applications, this approximation poses little difficulty: as we demonstrate in Sec. 5, ML model training gradients generally have a relatively narrow exponent range, and the workload is in any event resilient to small inaccuracies. For others, it may be more problematic; in those cases, the architectural modifications of Sec. 4.2 will be needed.

5 Case Study: Distributed ML Training

As the model and dataset sizes have increased for ML training jobs, large-scale distributed training has become increasingly important [1, 12, 13, 21, 33, 38, 40, 41, 47, 67, 81, 91, 113]. In this paper, we focus specifically on data-parallel training, a common approach to distributed training.[3] In data-parallel training, the dataset is partitioned to multiple worker machines, each with a replica of the model. In a training iteration, each machine performs learning on its local dataset and model, generating gradient vectors. These gradient vectors are then used to update the weights that make up the model. Modern supervised ML typically employs stochastic gradient descent (SGD) [82, 83, 95] or its variants as the optimizer for iterative training. In general, the core operation of SGD is as follows:

$$weight_{(next)} = weight_{(current)} - learning\_rate \cdot gradient_{(current)},$$

where $gradient_{(current)}$ is the element-wise mean of all the local gradient vectors produced by each worker. Computing this mean requires summing (or aggregating) gradient vectors from all workers.

Prior work has observed that, as the number of workers and the size of the model grows, communication costs – specifically, the gradient aggregation procedure – increasingly become a bottleneck in distributed training [70, 71, 98, 118]. Gradient aggregation can be viewed as an "all-reduce" collective operation, a familiar concept from the HPC world – the gradient vectors are **gathered** from all worker machines, **reduced** to one vector, and **sent back** to all worker machines. It is traditionally implemented either using a parameter server [67] or a distributed reduction protocol like ring all-reduce [6, 90].

In-network aggregation has been proposed as a promising way to accelerate this collective operation, and thus distributed training [2, 27, 31, 32, 57, 61, 70, 98]. In-network aggregation performs the "reduce" (*i.e.*, sum) step of all-reduce in a network switch on the fly. This offers higher throughput and lower latency than a parameter server approach, where both the network link and host-side network stack can become bottlenecks. Compared to ring-based and other distributed all-reduce algorithms, in-network aggregation requires exchanging fewer messages, again reducing latency and network usage.

PISA switches are well suited for, and have been used for, implementing in-network aggregation without specialized hardware. A major challenge, however, is the lack of floating point support. The recent state-of-the-art in-network aggregation work, SwitchML [98], works around this by quantizing floating point values at end hosts so that the PISA switch only operates on fixed-point values. While this quantization approach has been shown not to impact accuracy [98], we show that it harms performance. In particular, quantization and format conversion requires significant CPU overhead on the worker hosts. Computing the scaling factor to use for each block also requires

---

[3]Other parallel modes, like model-parallel, may also benefit from what is discussed in this work, but we do not explore them here.

an additional network round trip. Both costs could be avoided if the switch could operate on floating point values directly.

5.1 Setup

Environments. Given the hardware constraints of the current Tofino ASIC described in Sec. 4.1, we are not able to evaluate FPISA's applicability/benefit on the distributed ML training scenario entirely on a real system. Hence, we employ different evaluation approaches for different aspects of the process.

Specifically, to measure training accuracy and the impact of error, we write a C library that simulates gradient aggregation using a faithful implementation of the FPISA-A addition algorithm and integrate this C library into PyTorch [89] to train the models. We use the apex [85] PyTorch extension to evaluate both FP32 and FP16 floating point formats. Experiments and plots with this approach are labeled with "**[SIMULATION]**".

To analyze the numerical characteristics of the trained models' gradients and measure training throughput, we use an 8-machine cluster where each node is equipped with one NVIDIA P100 16 GB GPU, two 10-core Intel Xeon E5-2630v4 2.2 GHz CPUs, and 128 GB of DRAM with data served from local SSD storage. The cluster is networked at 100 Gbps and includes one Tofino-based Wedge100BF-65X programmable switch. This cluster deploys in-network aggregation through SwitchML [98].

For gradient numerical analysis, we directly dump the gradient vectors during the training processes. In these experiments, the workers compute gradients in the FP32 floating point format. Experiments and plots with this approach are labeled with "**[TRACE]**".

For performance (speedup) evaluation, we seek to measure the performance that FPISA-A can achieve with our variable-length shift extension, which allows multiple parallel FPISA-A instances per pipeline. Because current switch hardware does not support this, we emulate FPISA-A-enabled performance by removing the end-host format conversion/quantization at the workers and performing integer computations in place of floating point computations on the switch. While this emulation setup gives nonsensical output, it provides a realistic expectation of FPISA-A performance because: (1) under Tofino, data plane programs experience a switch processing latency that depends only on the number of stages and not on the computation intensity of their specific operations, without any effect on throughput (data plane programs operate at line rate) as confirmed experimentally in previous work (e.g., [17]); (2) SwitchML uses the full set of stages on the ingress pipelines of Tofino and any potential increase of in-switch latency can be mitigated by increasing the number of aggregation slots. Note that we use this approach only for performance evaluation, and it runs on the testbed configuration described above. Experiments and plots with this approach are labeled with "**[EMULATION]**".

Benchmarks. We select seven popular state-of-the-art ML models. These models are MobileNetV2 [96], VGG19 [101],
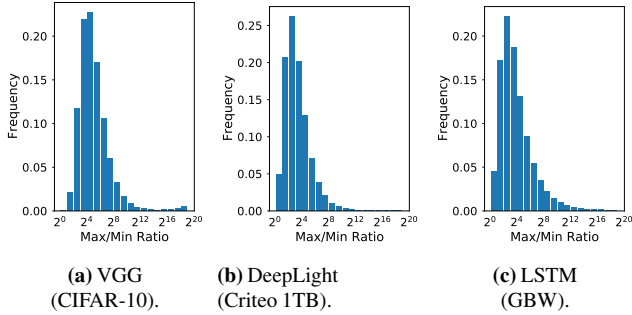
(a) VGG (CIFAR-10). **(b)** DeepLight (Criteo 1TB). **(c)** LSTM (GBW).

Figure 7: [TRACE] Element-wise max/min ratio distribution of different models (datasets).

(a) Epoch #1. **(b)** Epoch #20. **(c)** Epoch #40.

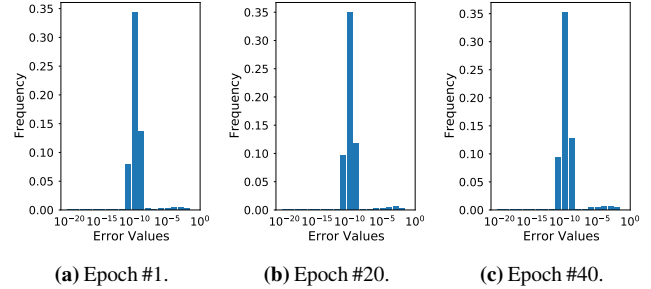**Figure 8:** [SIMULATION] FPISA-A's error distribution of VGG19 gradient aggregation at early, middle, and final training stages.

ResNet-50 [37], GoogleNet [108], LSTM [52], Deep-Light [22], and BERT [24]. We use all of these to evaluate training throughput, but evaluate accuracy only for the first four, since emulating FPISA-A in software is costly and those four CNNs train much faster than the other models. For CNN models, we use the CIFAR-10 dataset [59] with a learning rate of 0.1, momentum of 0.9, and weight decay of 0.0005. For other models, we use the same setting as in the SwitchML evaluation [98]. Regarding the batch size, for the accuracy experiments, we use a batch size of 16 because small batches represent a worst-case configuration from an accuracy standpoint; for the performance experiments, we use the standard batch sizes of each model listed in the MLPerf benchmark [80] and the SwitchML work [98] (*i.e.*, 2^{13} for DeepLight, 4 for BERT and 64 for others).

5.2 Characteristics of Training Gradients

The gradient aggregation workload has some common numerical characteristics that make it well suited for in-network aggregation with FPISA. In particular, FPISA can be used with existing Tofino switches using the FPISA-A approximation (Sec. 4.3); the resulting numerical error is rare and (as we demonstrate) has no impact on training accuracy.

High aggregation parallelism. In general, for each training iteration, the entire gradient vector corresponding to the training model needs to be aggregated from all worker machines. These vectors can range from several MBs to GBs. Aggregation is just vector addition; this element-wise summation provides ample parallelism.

Vector-wise distribution. As studied in INCEPTIONN [71], gradient values in each vector largely fall in the narrow range of $[-1, 1]$, and most are close to "0".

Element-wise distribution. We find that for the same element from different workers' gradient vectors at the same iteration, the relative range is also narrow. To demonstrate this, we analyze the distribution of element-wise *max/min* ratio among eight workers' gradient vectors of the training of three models and datasets (see Sec. 5.3 for detailed setup and configuration), and plot the results at the early training phase (*i.e.*, the first epoch) in Fig. 7 (we have observed similar distributions

through the mid/final phases of the training). We find that, regardless of the model and dataset, most ($\sim 83\%$) elements' *max/min* ratio is smaller than 2^7.

Precision loss/error tolerance. It is well known that small floating point error does not dramatically affect the convergence and final accuracy of ML models [15, 19, 23, 71]. This observation has motivated extensive prior research about training with low or mixed-precision floating point operations [19, 25, 46, 50, 79, 115] and compression or quantization [35, 39, 44, 71].

Thanks to these numerical characteristics, FPISA-A addition can be directly applied to the in-network aggregation scenario on current Tofino switches. As discussed in Sec. 4.3, the lack of a shift-and-add operation introduces error only when adding values that differ by more than a 2^7 ratio – which Fig. 7 shows is rare – and the workload can tolerate such error. We show later that it has no impact on model accuracy or convergence. However, as discussed in Sec. 4.1, the cost of shift operations does mean the current Tofino only accommodates one FPISA-A module per pipeline. Hence, in-network aggregation performance will benefit from the variable-length shift enhancement we propose.

5.3 Evaluation

We take a two-step approach to our evaluation. (1) We first show that FPISA-A addition will not affect the training convergence (*i.e.*, FPISA-A will not incur more training iterations), and do not consider time-wise performance. (2) We demonstrate that FPISA-A can reduce the time of each training iteration and do not consider the convergence (because it is agnostic to per-iteration time). Taken together, we conclude that FPISA-A reduces end-to-end training time.

5.3.1 FPISA-A Error Analysis

To investigate the errors to which FPISA-A addition may lead, we record the gradient vectors from eight workers during a training job. We use the C library to compare the results of FPISA-A vs. standard floating point addition for aggregating the same gradient vectors. Fig. 8 shows the (absolute) error distribution of VGG19 during different training phases.

Similar to the gradient distribution [71], the error distribu-

tion remains similar among early, middle, and final phases of training, showing FPISA-A's wide applicability. Most errors (>95%) are in the range of $[10^{-10}, 10^{-8}]$, enough to be tolerated by ML training, which we demonstrate in the next section. We further investigate the sources of the errors and find that most errors come from rounding, while the errors caused by the overwrite and left-shift mechanisms happen rarely (less than 0.9% and 0.1%, respectively, among all the addition operations in the aggregation procedure). These errors arise because, in some cases, a gradient vector's element-wise distribution is larger than FPISA-A's left-shift headroom. As a result, the smaller values may be ignored in the aggregation procedure, leading to small errors (*i.e.*, smaller than 10^{-8}).

Note that a switch implementing the full FPISA proposal, rather than just FPISA-A, would not experience these overwrite errors. Note also that no overflow occurs in this experiment, since the number of workers, and thus the number of operations per vector element, is less than the headroom available in the mantissa register.

5.3.2 FPISA-A's Impact on Training Convergence

We investigate whether FPISA-A will lead to training accuracy loss, due to the errors it imposes. We train four ML models for 40 epochs with both default and FPISA-A addition in gradient aggregation. To show FPISA-A's adaptability on different floating point formats, we train using both standard single-precision FP32 and half-precision FP16 for each model.

We plot the accuracy value during the training procedures of each model in Fig. 9 to observe FPISA-A's impact on convergence. Note that the jitters in the curves are due to the small batch size we are choosing; these are normal and do not affect the training procedure. First, we find that floating point precision does affect the training convergence. That is, in all four models, we observe slower convergence of FP16-based training compared to regular FP32-based training, as well as the final accuracy. However, FPISA-A's addition errors will not amplify such gaps. In most cases, the curve of FPISA-A addition is closely aligned with the curve of default addition. After 40 epochs, the accuracy differs by less than 0.1%. The results also demonstrate that regardless of the floating point format, FPISA-A addition will not degrade each model's accuracy. Hence, we argue that FPISA-A will not prolong the training by adding necessary epochs to converge.

5.3.3 Training Speedup with FPISA-A

In the next experiments, we evaluate the potential speedup of FPISA-A in an end-to-end training setting as well as the resulting reduction of host-based quantization overheads. SwitchML uses CPU cores at workers to scale and transform the numeric representation of gradient vectors, including both floating-point/integer conversion and byte order conversion. In contrast, FPISA-A does not have these overheads as it sends gradient vectors as floating point values directly. As described in Sec. 5.1, from an end-to-end perspective, this is the sole source of expected performance variation between SwitchML and FPISA-A. Thus, we vary the number of CPU cores and measure the throughput differences between these approaches through a microbenchmark.

In this microbenchmark, two workers reduce a 1 GB gradient vector;[4] we measure the time to complete the operation across the workers. We use 256 element packets which is the largest that SwitchML supports. After 50 warm-up invocations, we perform 250 reductions and report median and 10th-90th percentiles as the error bars.

SwitchML baselines. We use SwitchML's RDMA transport since it is more efficient that the DPDK one, and we run two versions to explore the performance implications of scaling and transforming gradient vectors on either the CPU or the GPU (where gradients are initially computed and stored). The base SwitchML version – denoted SwitchML/CPU – uses CPU cores. This benchmark assumes that the gradient vectors are already in host memory. Further, we create a new version of SwitchML – denoted SwitchML/GPU – that uses the GPU to scale and transform gradient vectors to the integer representation before copying them to pinned host memory.

Recall that SwitchML scales the gradient vectors in chunks, using a scaling factor adapted to each chunk based on a maximum exponent calculation that involves a round trip over the network. SwitchML saves the maximum exponent calculation's network overhead by overlapping the aggregation of the current chunk with the exponent calculation of the next chunk.

For SwitchML/CPU, we keep the original SwitchML logic where one chunk is equivalent to the RDMA message size. For SwitchML/GPU, we use a separate CUDA stream for each CPU core to allow parallel kernel execution. We also introduce a performance optimization where we asynchronously de-quantize aggregated messages from integer into floating point values on a separate CUDA stream thus having two CUDA streams for each CPU core. Despite these optimizations, there is an inherent overhead with launching one GPU kernel for each chunk. One potential way to avoid this could be to execute the per-chunk maximum exponent calculation as a pre-processing operation before the in-network aggregation phase; we leave this to future work.

FPISA-A approaches. We run our FPISA-A emulation in three settings. (1) FPISA-A/CPU directly adopts the RDMA implementation of SwitchML and disables host-based type conversions. SwitchML's RDMA implementation, however, involves a CPU memory copy operation into a staging area. This memory copy is not necessary in the case of FPISA-A since it can operate entirely on memory-resident native FP vectors without quantization; thus, we include a further optimization – (2) FPISA-A/CPU(Opt) – that omits this extra memory copy. Lastly, (3) FPISA-A/GPU (for comparison

---

[4]We use two workers to exclude the synchronization variability among a larger number of workers. This is to better quantify the performance differences due to the scaling and transformation overheads. We also tried 100 MB with similar results.
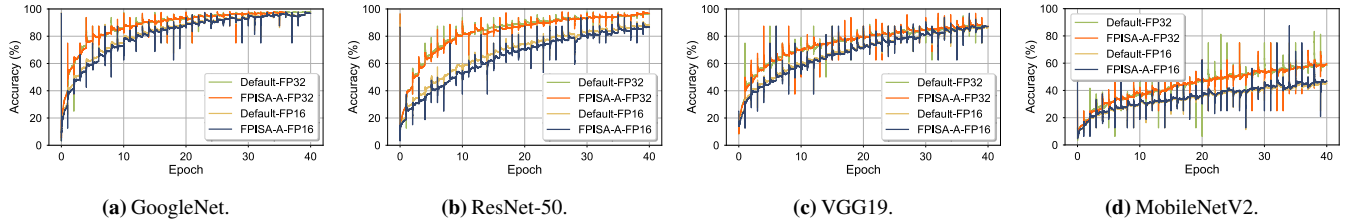
(a) GoogleNet. **(b)** ResNet-50. **(c)** VGG19. **(d)** MobileNetV2.

Figure 9: [SIMULATION] Accuracy curves of different ML models with default addition and FPISA-A addition.

Cores vs Goodput (CPU) (16KB Msgs) Message Size vs Goodput (CPU) (4 Cores) Message Size vs Goodput (GPU) (4 Cores)
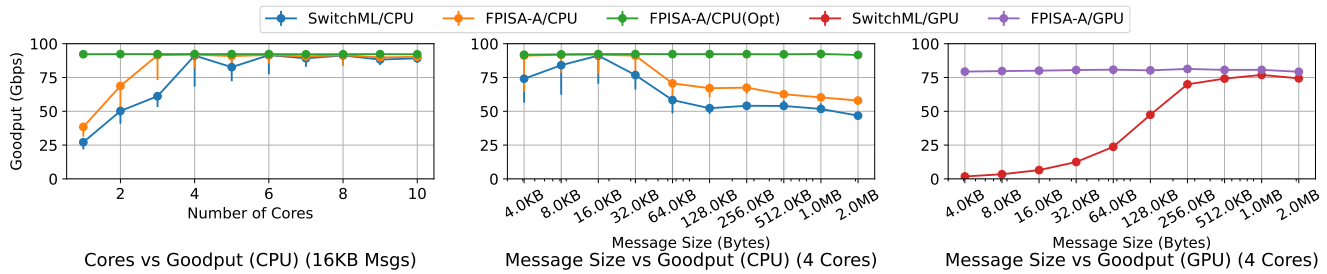
Figure 10: [EMULATION] Goodput of different floating point approaches on microbenchmark. The maximum theoretical goodput with framing overhead is 92 Gbps.
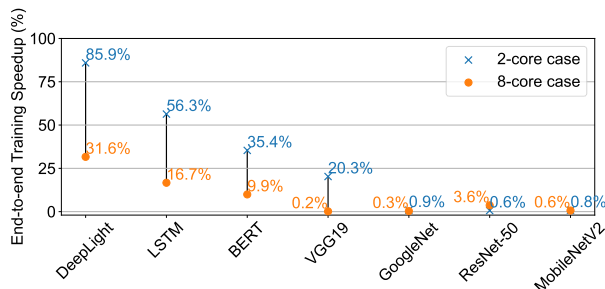

Figure 11: [EMULATION] End-to-end training time speedup of FPISA-A compared to the default SwitchML.

against SwitchML/GPU) includes a copy from GPU memory to pinned host memory and back.[5]

Because FPISA-A operates directly on FP vectors, we introduce two performance optimizations for FPISA-A/GPU that are not applicable to SwitchML/GPU (due to the need for chunk-based quantization). First, we use batching to amortize the cost of launching one copy operation for each chunk. Second, we asynchronously copy from GPU to host memory as a pipeline of one batch ahead of what needs to be consumed. Further, similar to the SwitchML/GPU case, we asynchronously copy back from host to GPU memory on a separate CUDA stream.

In-network aggregation goodput. Fig. 10 (left) shows that FPISA-A/CPU requires three CPU cores to achieve the 92 Gbps maximum goodput, as opposed to SwitchML/CPU, which needs four cores.[6] FPISA-A/CPU(Opt) achieves the maximum goodput with just a single core. This leaves more CPU cycles for data I/O, potentially avoiding training job stalls while waiting for input data to be preprocessed.

Figure 12: [EMULATION] SwitchML/GPU overheads at each iteration of the microbenchmark. To achieve high goodput, a message size of 256 KB or beyond is necessary. At smaller message sizes, the kernel and copy launches (solid lines) introduce a substantial latency compared to the actual kernel execution or copy latency (dashed lines).

The message size for this benchmark is 16 KB, which allows SwitchML/CPU to reach peak performance, according to the SwitchML paper [98]. Fig. 10 (middle) illustrates that FPISA-A achieves maximum goodput for a wide range of message sizes.

For the GPU variants, we find that the message/chunk size is the most important factor. Fig. 10 (right) shows that SwitchML/GPU is inefficient with message sizes below 256 KB. This is due to overheads of GPU kernel launches and copies at small message sizes (cf. Fig. 12). Increasing the number of cores does not help because CUDA implicitly synchronizes all kernel launch calls (kernel execution can be parallelized whereas kernel launches cannot). In contrast, using just a single CPU core, FPISA-A/GPU achieves the best possible performance – limited to 80 Gbps only by the bidirectional copy bandwidth of the GPU copy engines – since it can copy chunks in larger batches.[7] We expect that without

---

[5]Our testbed does not support GPU Direct, which would enable FPISA-A to use RDMA transfers out of and into GPU memory.

[6]SwitchML/CPU with 5 cores has a small performance dip due to work imbalance across cores in this particular configuration.

[7]We copy memory using 1 MB chunks as it gives the best results irrespective of the RDMA message size.

---

this bidirectional copy bandwidth limit (a constraint of our environment), FPISA-A/GPU would match the performance of FPISA-A/CPU(Opt) since it completely overlaps the memory copying with CPU and network operations.

SwitchML/GPU with a chunk size of 1 MB reaches a performance comparable (but still below) to FPISA-A/GPU. However, this requires an equally large RDMA message size whereas FPISA-A/GPU performs well even with 4 KB messages. Using large message sizes has several negative implications. First, it can introduce larger errors in SwitchML's quantization scheme since it chooses the scaling factor from a larger chunk. Second, it hurts the performance of loss recovery because the loss of a single packet entails resending the entire 1 MB message (1024 packets). Third, the performance degrades past a certain message size. This is due to limited network capacity and the reduction of pipelining, which in turn reduces the performance benefits of SwitchML's streaming aggregation. Thus, we conclude that, although performing quantization on the GPU might still be an interesting possibility for SwitchML, more work is necessary to devise an efficient implementation without increasing quantization errors and without affecting the GPU's availability for training.

Training throughput. We now confirm that FPISA-A's benefits translate into higher end-to-end training throughput. Fig. 11 reports the training throughput for seven real-world DNN benchmarks. For these experiments, we restrict the comparison to the DPDK implementation because SwitchML/RDMA is not currently integrated into the ML frameworks [98]. We focus on two scenarios – using either two or eight cores – and we measure the speedup in terms of training throughput (samples/s). We observe that FPISA-A speeds up training by up to 85.9% and 31.6% for the 2-core case and the 8-core case, respectively. Importantly, the higher speedup factors are obtained when using just two cores for communication, which frees up six cores for data I/O in this setting. The speedup is particularly significant in communication-bottlenecked models (*e.g.*, DeepLight, LSTM, BERT, VGG19), where FPISA-A is up to 85.9% faster compared to SwitchML when using the same number of cores. On the other hand, we do not see significant benefits of FPISA-A on models like GoogleNet, MobileNetV2, and ResNet-50, which are compute-bottlenecked.

By combining the accuracy results and the per-iteration end-to-end results, we can conclude that FPISA-A is able to reduce the end-to-end training time of a wide range of ML models.

6 Related Work

Accelerating distributed/networking applications with programmable switches. Recently, programmable switches have been used to accelerate a broad range of applications, including distributed key-value stores [49, 66, 112], distributed transactions [48, 64, 117], distributed storage [72, 120], packet queuing/scheduling [100, 103], network functions [56, 78], and network telemetry [7, 34, 105, 119]. While most of them

deal with packet header processing with few arithmetic operations, some perform computation on the packet's payload. SwitchML [98] and ATP [61] leverage switches for gradient aggregation but are constrained to fixed-point aggregation, which may lead to costly format conversion on the end-host and additional network round trips for exponent communication.

FPISA's approach is also applicable to other applications involving floating point operations and in-switch computing. For example, NETACCEL [63] and Cheetah [110] propose to use programmable switches to accelerate database queries by data pruning or query offloading. With the proposed architecture enhancements, FPISA can accelerate such queries with floating point as datatype. Also, other more complex floating point operations may be needed for future applications (*e.g.*, congestion control [26, 54] and network security [34]). Sec. 3.3 briefly discusses the possibility of supporting them.

Resource allocation. Much research has studied how to use in-network rate computations to support congestion control (e.g., XCP [54] and RCP [26]), queue management (e.g., CoDel [84] and AIFO [116]), or load balancing (e.g., CONGA [4]). P4QCN [29], P4-CoDel [60], and P4-ABC [77] are P4 implementations of specific protocols that require floating point support – currently unavailable in switch hardware. Sharma *et al*. proposed a library that applies approximation to work around this limitation [99]. InREC [51] and NetFC [16] proposed to use table-lookup for floating point operation emulation in programmable switches. However, they are constrained to stateless operations and need extra RAM space to store the tables. Also, few floating point operations can be done per packet, limiting parallelism. FPISA may enable new design options for in-switch resource allocation.

Extending switches' processing capability. Proposed enhancements to the RMT architecture [9] include transactions [102], disaggregated memory [14], and better stateful data plane support [28]. While many focus on improving stateful computations, none address floating point operations.

7 Conclusion

In this work, we propose FPISA, a floating point representation designed to work efficiently in programmable switches. We first implement FPISA on a commodity Intel Tofino switch, but its design limits throughput and accuracy. We then propose hardware changes based on the Banzai programmable switch architecture to avoid these limitations. We demonstrate their feasibility through synthesis using a 15-nm standard-cell library, and find minimal impact on area, power, and timing. Finally, we investigate the benefit of FPISA by implementing accelerators for distributed training application, evaluating its performance on a switch implementing our changes using emulation. We find that FPISA allows distributed training to use 25-75% fewer CPU cores and provide up to 85.9% better throughput in a CPU-constrained environment than the state-of-the-art framework.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, Savannah, GA, Nov. 2016.

[2] N. Adiga, G. Almasi, G. Almasi, Y. Aridor, R. Barik, D. Beece, R. Bellofatto, G. Bhanot, R. Bickford, M. Blumrich, A. Bright, J. Brunheroto, C. Caşcaval, J. Castaños, W. Chan, L. Ceze, P. Coteus, S. Chatterjee, D. Chen, G. Chiu, T. Cipolla, P. Crumley, K. Desai, A. Deutsch, T. Domany, M. Dombrowa, W. Donath, M. Eleftheriou, C. Erway, J. Esch, B. Fitch, J. Gagliano, A. Gara, R. Garg, R. Germain, M. Giampapa, B. Gopalsamy, J. Gunnels, M. Gupta, F. Gustavson, S. Hall, R. Haring, D. Heidel, P. Heidelberger, L. Herger, D. Hoenicke, R. Jackson, T. Jamal-Eddine, G. Kopcsay, E. Krevat, M. Kurhekar, A. Lanzetta, D. Lieber, L. Liu, M. Lu, M. Mendell, A. Misra, Y. Moatti, L. Mok, J. Moreira, B. Nathanson, M. Newton, M. Ohmacht, A. Oliner, V. Pandit, R. Pudota, R. Rand, R. Regan, B. Rubin, A. Ruehli, S. Rus, R. Sahoo, A. Sanomiya, E. Schenfeld, M. Sharma, E. Shmueli, S. Singh, P. Song, V. Srinivasan, B. Steinmacher-Burow, K. Strauss, C. Surovic, R. Swetz, T. Takken, R. Tremaine, M. Tsao, A. Umamaheshwaran, P. Verma, P. Vranas, T. Ward, M. Wazlowski, W. Barrett, C. Engel, B. Drehmel, B. Hilgart, D. Hill, F. Kasemkhani, D. Krolak, C. Li, T. Liebsch, J. Marcella, A. Muff, A. Okomo, M. Rouse, A. Schram, M. Tubbs, G. Ulsh, C. Wait, J. Wittrup, M. Bae, K. Dockser, L. Kissel, M. Seager, J. Vetter, and K. Yates. An overview of the BlueGene/L supercomputer. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing (SC'02)*, Baltimorem, MD, Nov. 2002.

[3] N. Alachiotis and A. Stamatakis. Efficient floating-point logarithm unit for FPGAs. In *Proceedings of the 2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW'10)*, Atlanta, GA, May 2010.

[4] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese. CONGA: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM SIGCOMM Conference (SIGCOMM'14)*, Chicago, IL, Aug. 2014.

[5] Arista. 7130 FPGA-enabled Network Switches. https://www.arista.com/en/products/7130-fpga-enabled-network-switches-quick-look, accessed in 2021.

[6] M. Barnett, L. Shuler, R. van De Geijn, S. Gupta, D. G. Payne, and J. Watts. Interprocessor collective communication library (InterCom). In *Proceedings of the 1994 IEEE Scalable High Performance Computing Conference (SHPCC'94)*, Knoxville, TN, May 1994.

[7] R. Ben Basat, S. Ramanathan, Y. Li, G. Antichi, M. Yu, and M. Mitzenmacher. PINT: Probabilistic in-band network telemetry. In *Proceedings of the 2020 ACM SIGCOMM Conference (SIGCOMM'20)*, Virtual Event, Aug. 2020.

[8] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3), 2014.

[9] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *Proceedings of the 2013 ACM SIGCOMM Conference (SIGCOMM'13)*, Hong Kong, China, Aug. 2013.

[10] Broadcom. NPL: Open, High-Level language for developing feature-rich solutions for programmable networking platforms. https://nplang.org/, accessed in 2021.

[11] Broadcom. Trident4 BCM56880 Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56880-series, accessed in 2021.

[12] Y. Chen, Y. Peng, Y. Bao, C. Wu, Y. Zhu, and C. Guo. Elastic parameter server load distribution in deep learning clusters. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC'20)*, Virtual Event, Oct. 2020.

[13] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project Adam: Building an efficient and scalable deep learning training system. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, Broomfield, CO, Oct. 2014.

[14] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Vargaftik, A. Berger, G. Mendelson, M. Alizadeh, S.-T. Chuang, I. Keslassy, A. Orda, and T. Edsall. dRMT: Disaggregated programmable switching. In *Proceedings of the 2017 ACM SIGCOMM Conference (SIGCOMM'17)*, Los Angeles, CA, Aug. 2017.

[15] M. Courbariaux, Y. Bengio, and J.-P. David. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024*, 2014.

[16] P. Cui, H. Pan, Z. Li, J. Wu, S. Zhang, X. Yang, H. Guan, and G. Xie. NetFC: Enabling accurate floating-point arithmetic on programmable switches. In *Proceedings of the 29th IEEE International Conference on Network Protocols*, Virtual Event, Nov. 2021.

[17] H. T. Dang, P. Bressana, H. Wang, K. S. Lee, N. Zilberman, H. Weatherspoon, M. Canini, F. Pedone, and R. Soulé. P4xos: Consensus as a network service. *IEEE/ACM Transactions on Networking*, 28(4), Aug. 2020.

[18] B. Darvish Rouhani, D. Lo, R. Zhao, M. Liu, J. Fowers, K. Ovtcharov, A. Vinogradsky, S. Massengill, L. Yang, R. Bittner, A. Forin, H. Zhu, T. Na, P. Patel, S. Che, L. C. Koppaka, X. Song, S. Som, K. Das, S. Tiwary, S. Reinhardt, S. Lanka, E. Chung, and D. Burger. Pushing the limits of narrow precision inferencing at cloud scale with microsoft floating point. In *Advances in neural information processing systems 33 (NeurIPS'20)*, Virtual Event, Dec. 2020.

[19] C. De Sa, M. Leszczynski, J. Zhang, A. Marzoev, C. R. Aberger, K. Olukotun, and C. Ré. High-accuracy low-precision training. *arXiv preprint arXiv:1803.03383*, 2018.

[20] D. De Sensi, S. Di Girolamo, S. Ashkboos, S. Li, and T. Hoefler. Flare: Flexible in-network allreduce. *arXiv preprint arXiv:2106.15565*, 2021.

[21] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng. Large scale distributed deep networks. In *Advances in neural information processing systems 25 (NIPS'12)*, Lake Tahoe, NV, Dec. 2012.

[22] W. Deng, J. Pan, T. Zhou, D. Kong, A. Flores, and G. Lin. DeepLight: Deep lightweight feature interactions for accelerating CTR predictions in ad serving. *arXiv preprint arXiv:2002.06987*, 2020.

[23] A. Devarakonda, M. Naumov, and M. Garland. Adabatch: Adaptive batch sizes for training deep neural networks. *arXiv preprint arXiv:1712.02029*, 2017.

[24] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[25] M. Drumond, T. LIN, M. Jaggi, and B. Falsafi. Training dnns with hybrid block floating point. In *Advances in Neural Information Processing Systems 31 (NeurIPS'18)*, Montreal, Canada, Dec. 2018.

[26] N. Dukkipati. *Rate Control Protocol (RCP): Congestion control to make flows complete quickly*. PhD thesis, Stanford University, Dept. of Electrical Engineering, 2007.

[27] N. Gebara, P. Costa, and M. Ghobadi. In-network aggregation for shared machine learning clusters. In *Proceedings of the 4th MLSys confrence (MLSys'21)*, Virtual Event, Apr. 2021.

[28] N. Gebara, A. Lerner, M. Yang, M. Yu, P. Costa, and M. Ghobadi. Challenging the stateless quo of programmable switches. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks (HotNets'20)*, Virtual Event, Nov. 2020.

[29] J. Geng, J. Yan, and Y. Zhang. P4QCN: Congestion control using P4-capable device in data center networks. *Electronics*, 8(3), 2019.

[30] Google Cloud. Using bfloat16 with TensorFlow models. https://cloud.google.com/tpu/docs/bfloat16, accessed in 2021.

[31] R. L. Graham, D. Bureddy, P. Lui, H. Rosenstock, G. Shainer, G. Bloch, D. Goldenerg, M. Dubman, S. Kotchubievsky, V. Koushnir, L. Levi, A. Margolin, T. Ronen, A. Shpiner, O. Wertheim, and E. Zahavi. Scalable hierarchical aggregation protocol (SHArP): A hardware architecture for efficient data reduction. In *Proceedings of the 1st Workshop on Optimization of Communication in HPC (COM-HPC'16)*, Salt Lake City, Utah, Nov. 2016.

[32] R. L. Graham, L. Levi, D. Burredy, G. Bloch, G. Shainer, D. Cho, G. Elias, D. Klein, J. Ladd, O. Maor, A. Marelli, V. Petrov, E. Romlet, Y. Qin, and I. Zemah. Scalable hierarchical aggregation and reduction protocol (SHARP) streaming-aggregation hardware design and evaluation. In *Proceedings of the 35th International Conference on High Performance Computing (ISC'20)*, Frankfurt/Main, Germany, June 2020.

[33] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, and C. Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19)*, Boston, MA, Feb. 2019.

[34] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 ACM SIGCOMM Conference (SIGCOMM'18)*, Budapest, Hungary, Aug. 2018.

[35] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[36] F. Hauser, M. Häberle, D. Merling, S. Lindner, V. Gurevich, F. Zeiger, R. Frank, and M. Menth. A survey on data plane programming with P4: Fundamentals, advances, and applied research, 2021.

[37] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the 2016 IEEE conference on computer vision and pattern recognition (CVPR'16)*, Las Vegas, NV, June 2016.

[38] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing. More effective distributed ML via a stale synchronous parallel parameter server. In *Advances in neural information processing systems 26 (NIPS'13)*, Lake Tahoe, NV, Dec. 2013.

[39] S. Horvath, C.-Y. Ho, L. Horvath, A. N. Sahu, M. Canini, and P. Richtarik. Natural compression for distributed deep learning. *arXiv preprint arXiv:1905.10988*, 2019.

[40] C. Hwang, T. Kim, S. Kim, J. Shin, and K. Park. Elastic resource sharing for distributed deep learning. In *Proceedings*

of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI'21), Virtual Event, Apr. 2021.

[41] F. N. Iandola, M. W. Moskewicz, K. Ashraf, and K. Keutzer. FireCaffe: Near-linear acceleration of deep neural network training on compute clusters. In *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR'16)*, Las Vegas, NV, June 2016.

[42] Intel Corporation. Intel Tofino. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html, accessed in 2021.

[43] Intel Corporation. Intel Tofino2. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html, accessed in 2021.

[44] A. Jain, A. Phanishayee, J. Mars, L. Tang, and G. Pekhimenko. Gist: Efficient data encoding for deep neural network training. In *Proceedings of the 45th International Symposium on Computer Architecture (ISCA'18)*, Los Angeles, CA, June 2018.

[45] T. Jepsen, L. P. de Sousa, M. Moshref, F. Pedone, and R. Soulé. Infinite resources for optimistic concurrency control. In *Proceedings of the ACM SIGCOMM 2018 Workshop on In-Network Computing (NetCompute'18)*, Budapest, Hungary, Aug. 2018.

[46] X. Jia, S. Song, W. He, Y. Wang, H. Rong, F. Zhou, L. Xie, Z. Guo, Y. Yang, L. Yu, T. Chen, G. Hu, S. Shi, and X. Chu. Highly scalable deep learning training system with mixed-precision: Training ImageNet in four minutes. *arXiv preprint arXiv:1807.11205*, 2018.

[47] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo. A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, Virtual Event, Nov. 2020.

[48] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica. NetChain: Scale-free sub-RTT coordination. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*, Renton, WA, Apr. 2018.

[49] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. NetCache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP'17)*, Shanghai, China, Oct. 2017.

[50] J. Johnson. Rethinking floating point for deep learning. *arXiv preprint arXiv:1811.01721*, 2018.

[51] M. Jose, K. Lazri, J. François, and O. Festor. InREC: In-network real number computation. In *Proceedings of the 2021 IFIP/IEEE International Symposium on Integrated Network Management*, Virtual Event, May 2021.

[52] R. Jozefowicz, O. Vinyals, M. Schuster, N. Shazeer, and Y. Wu. Exploring the limits of language modeling. *arXiv preprint arXiv:1602.02410*, 2016.

[53] D. D. Kalamkar, D. Mudigere, N. Mellempudi, D. Das, K. Banerjee, S. Avancha, D. T. Vooturi, N. Jammalamadaka, J. Huang, H. Yuen, J. Yang, J. Park, A. Heinecke, E. Georganas, S. Srinivasan, A. Kundu, M. Smelyanskiy, B. Kaul, and P. Dubey. A study of bfloat16 for deep learning training. *arXiv preprint arXiv:1905.12322*, 2019.

[54] D. Katabi, M. Handley, and C. Rohrs. Congestion control for high bandwidth-delay product networks. *ACM SIGCOMM Computer Communication Review*, 32(4), 2002.

[55] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford. HULA: Scalable load balancing using programmable data planes. In *Proceedings of the 2016 Symposium on SDN Research (SOSR'16)*, Santa Clara, CA, Mar. 2016.

[56] D. Kim, Z. Liu, Y. Zhu, C. Kim, J. Lee, V. Sekar, and S. Seshan. TEA: Enabling state-intensive network functions on programmable switches. In *Proceedings of the 2020 ACM SIGCOMM Conference (SIGCOMM'20)*, Virtual Event, Aug. 2020.

[57] B. Klenk, N. Jiang, G. Thorson, and L. Dennison. An in-network architecture for accelerating shared-memory multiprocessor collectives. In *Proceedings of the 47th International Symposium on Computer Architecture (ISCA'20)*, Virtual Event, May 2020.

[58] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1), 2014.

[59] A. Krizhevsky. The CIFAR-10 dataset. https://www.cs.toronto.edu/~kriz/cifar.html, accessed in 2021.

[60] R. Kundel, J. Blendin, T. Viernickel, B. Koldehofe, and R. Steinmetz. P4-CoDel: Active queue management in programmable data planes. In *Proceedings of 2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, Verona, Italy, Nov. 2018.

[61] C. Lao, Y. Le, K. Mahajan, Y. Chen, W. Wu, A. Akella, and M. Swift. ATP: In-network aggregation for multi-tenant learning. In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI'21)*, Virtual Event, Apr. 2021.

[62] A. S. Leon, K. W. Tam, J. L. Shin, D. Weisner, and F. Schumacher. A power-efficient high-throughput 32-thread SPARC processor. *IEEE Journal of Solid-State Circuits*, 42(1), 2007.

[63] A. Lerner, R. Hussein, and P. Cudre-Mauroux. The case for network accelerated query processing. In *Proceedings of the 9th Biennial Conference on Innovative Data Systems Research (CIDR'19)*, Asilomar, CA, Jan. 2019.

[64] J. Li, E. Michael, and D. R. K. Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP'17)*, Shanghai, China, Oct. 2017.

[65] J. Li, E. Michael, A. Szekeres, N. K. Sharma, and D. R. K. Ports. Just say NO to Paxos overhead: Replacing consensus with network ordering. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, Savannah, GA, Nov. 2016.

[66] J. Li, J. Nelson, E. Michael, X. Jin, and D. R. K. Ports. Pegasus: Tolerating skewed workloads in distributed storage with in-network coherence directories. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, Virtual Event, Nov. 2020.

[67] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, Broomfield, CO, Oct. 2014.

[68] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09)*, New York, NY, Dec. 2009.

[69] Y. Li and W. Chu. Implementation of single precision floating point square root on FPGAs. In *Proceedings of the 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'97)*, Apr. 1997.

[70] Y. Li, I.-J. Liu, Y. Yuan, D. Chen, A. Schwing, and J. Huang. Accelerating distributed reinforcement learning with in-switch computing. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA'19)*, Phoenix, AZ, June 2019.

[71] Y. Li, J. Park, M. Alian, Y. Yuan, Z. Qu, P. Pan, R. Wang, A. G. Schwing, H. Esmaeilzadeh, and N. S. Kim. A network-centric hardware/algorithm co-design to accelerate distributed training of deep neural networks. In *Proceedings of the 51st International Symposium on Microarchitecture (MICRO'18)*, Fukuoka, Japan, Oct. 2018.

[72] Z. Liu, Z. Bai, Z. Liu, X. Li, C. Kim, V. Braverman, X. Jin, and I. Stoica. DistCache: Provable load balancing for large-scale storage systems with distributed caching. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*, Boston, MA, Feb. 2019.

[73] M. Martins, J. M. Matos, R. P. Ribas, A. Reis, G. Schlinker, L. Rech, and J. Michelsen. Open cell library in 15nm FreePDK technology. In *Proceedings of the 2015 Symposium on International Symposium on Physical Design (ISPD'15)*, Monterey, CA, Mar. 2015.

[74] S. Mathew, M. Anders, B. Bloechel, T. Nguyen, R. K. Krishnamurthy, and S. Borkar. A 4-GHz 300-mW 64-bit integer execution ALU with dual supply voltages in 90-nm CMOS. *IEEE Journal of Solid-State Circuits*, 40(1), 2005.

[75] Mellanox. Mellanox scalable hierarchical aggregation and reduction protocol (SHARP). http://www.mellanox.com/page/products_dyn?product_family=261&mtag=sharp, accessed in 2021.

[76] Mellanox. QM8700 Mellanox Quantum HDR Edge Switch. https://www.mellanox.com/files/related-docs/prod_ib_switch_systems/PB_QM8700.pdf, accessed in 2021.

[77] M. Menth, H. Mostafaei, D. Merling, and M. Häberle. Implementation and evaluation of activity-based congestion management using P4 (P4-ABC). *Future Internet*, 11, 2019.

[78] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. SilkRoad: Making stateful layer-4 load balancing fast and cheap using switching ASICs. In *Proceedings of the 2017 ACM SIGCOMM Conference (SIGCOMM'17)*, Los Angeles, CA, Aug. 2017.

[79] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.

[80] MLCommons. Mlperf benchmark. https://mlcommons.org/en/training-normal-10/, accessed in 2021.

[81] P. Moritz, R. Nishihara, I. Stoica, and M. I. Jordan. SparkNet: Training deep networks in Spark. *arXiv preprint arXiv:1511.06051*, 2015.

[82] A. Nemirovski, A. Juditsky, G. Lan, and A. Shapiro. Robust stochastic approximation approach to stochastic programming. *SIAM Journal on optimization*, 19(4), 2009.

[83] A. S. Nemirovsky and D. B. Yudin. *Problem complexity and method efficiency in optimization*. Society for Industrial and Applied Mathematics, 1983.

[84] K. Nichols, V. Jacobson, A. McGregor, and J. Iyengar. Controlled delay active queue management. RFC 8289, 2018. https://tools.ietf.org/html/rfc8289.

[85] NVIDIA. apex: Tools for easy mixed precision and distributed training in Pytorch. https://github.com/NVIDIA/apex, accessed in 2021.

[86] NVIDIA blog. TensorFloat-32 in the A100 GPU accelerates AI training, HPC up to 20x. https://blogs.nvidia.com/blog/2020/05/14/tensorfloat-32-precision-format/, accessed in 2021.

[87] S. Oberman. Floating point division and square root algorithms and implementation in the AMD-K7 microprocessor. In *Proceedings of the 14th IEEE Symposium on Computer Arithmetic*, Adelaide, Australia, Apr. 1999.

[88] OpenSwitch. Cavium-XPliant family of programmable ethernet switches. https://www.openswitch.net/cavium/, accessed in 2021.

[89] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems 32 (NIPS'19)*, Vancouver, Canada, Dec. 2019.

[90] P. Patarasuk and X. Yuan. Bandwidth Optimal All-reduce Algorithms for Clusters of Workstations. *Journal of Parallel and Distributed Computing*, 69(2), 2009.

[91] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo. A generic communication scheduler for distributed DNN training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)*, Huntsville, Canada, Oct. 2019.

[92] Y. Piasetzky, M. Kadosh, M. Pritsak, O. Shabtai, A. Lo, and G. Lu. Switch asic programmability in hybrid mode. In *Proceedings of 2018 IEEE 26th International Conference on Network Protocols (ICNP'18)*, Cambridge, UK, Sept. 2018.

[93] D. R. K. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy. Designing distributed systems using approximate synchrony in datacenter networks. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*, Oakland, CA, May 2015.

[94] D. R. K. Ports and J. Nelson. When should the network be the computer? In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS'19)*, Bertinoro, Italy, May 2019.

[95] H. Robbins and S. Monro. A stochastic approximation method. *The annals of mathematical statistics*, 1951.

[96] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. MobileNetV2: Inverted residuals and linear bottlenecks. In *Proceedings of the 2018 IEEE conference on computer vision and pattern recognition (CVPR'18)*, Salt Lake City, UT, June 2018.

[97] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis. In-network computation is a dumb idea whose time has come. In *Proceedings of the 16th Workshop on Hot Topics in Networks (HotNets'17)*, Palo Alto, CA, Nov. 2017.

[98] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. R. Ports, and P. Richtárik. Scaling distributed machine learning with in-network aggregation. In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI'21)*, Virtual Event, Apr. 2021.

[99] N. K. Sharma, A. Kaufmann, T. Anderson, A. Krishnamurthy, J. Nelson, and S. Peter. Evaluating the power of flexible packet processing for network resource allocation. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*, Boston, MA, Mar. 2017.

[100] N. K. Sharma, M. Liu, K. Atreya, and A. Krishnamurthy. Approximating fair queueing on reconfigurable switches. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*, Renton, WA, Apr. 2018.

[101] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[102] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM'16)*, Florianopolis, Brazil, Aug. 2016.

[103] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown. Programmable packet scheduling at line rate. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM'16)*, Florianopolis, Brazil, Aug. 2016.

[104] P. Soderquist and M. Leeser. Area and performance tradeoffs in floating-point divide and square-root implementations. *ACM Computing Surveys*, 28(3), 1996.

[105] J. Sonchack, A. J. Aviv, E. Keller, and J. M. Smith. Turboflow: Information rich flow record generation on commodity switches. In *Proceedings of the 13th European Conference on Computer Systems (EuroSys'18)*, Porto, Portugal, Apr. 2018.

[106] A. Svyatkovskiy, J. Kates-Harbeck, and W. Tang. Training distributed deep recurrent neural networks with mixed precision on GPU clusters. In *Proceedings of the Machine Learning on HPC Environments (MLHPC'17)*, Denver, CO, Nov. 2017.

[107] Synopsys. Design Compiler Graphical. https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/design-compiler-graphical.html, accessed in 2021.

[108] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Proceedings of the 2015 IEEE conference on computer vision and pattern recognition (CVPR'15)*, Boston, MA, June 2015.

[109] P.-T. P. Tang. Table-driven implementation of the logarithm function in IEEE floating-point arithmetic. *ACM Transactions on Mathematical Software*, 16(4), 1990.

[110] M. Tirmazi, R. Ben Basat, J. Gao, and M. Yu. Cheetah: Accelerating database queries with switch pruning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20)* https://arxiv.org/pdf/2004.05076.pdf, Virtual Event, June 2020.

[111] Y. Tokusashi, H. T. Dang, F. Pedone, R. Soulé, and N. Zilberman. The case for in-network computing on demand. In *Proceedings of the 14th EuroSys Conference (EuroSys'19)*, Dresden, Germany, Mar. 2019.

[112] Y. Tokusashi, H. Matsutani, and N. Zilberman. LaKe: An energy efficient, low latency, accelerated key-value store. *arXiv preprint arXiv:1805.11344*, 2018.

[113] S. Venkataraman, E. Bodzsar, I. Roy, A. AuYoung, and R. S. Schreiber. Presto: Distributed machine learning and graph processing with sparse matrices. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys'13)*, Prague, Czech Republic, Apr. 2013.

[114] M. Voogel, Y. Frans, and M. Ouellette. Xilinx Versal Premium series. In *HotChips'20*, Virtual Event, Aug. 2020.

[115] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan. Training deep neural networks with 8-bit floating point numbers. In *Advances in neural information processing systems 31 (NIPS'18)*, Montreal, Canada, Dec. 2018.

[116] Z. Yu, C. Hu, J. Wu, X. Sun, V. Braverman, M. Chowdhury, Z. Liu, and X. Jin. Programmable packet scheduling with a single queue. In *Proceedings of the 2021 ACM SIGCOMM Conference (SIGCOMM'21)*, Virtual Event, Aug. 2021.

[117] Z. Yu, Y. Zhang, V. Braverman, M. Chowdhury, and X. Jin. NetLock: Fast, centralized lock management using programmable switches. In *Proceedings of the 2020 ACM SIGCOMM Conference (SIGCOMM'20)*, Virtual Event, Aug. 2020.

[118] Z. Zhang, C. Chang, H. Lin, Y. Wang, R. Arora, and X. Jin. Is network the bottleneck of distributed training? In *Proceedings of the Workshop on Network Meets AI & ML (NetAI'20)*, Virtual Event, Aug. 2020.

[119] Y. Zhou, C. Sun, H. H. Liu, R. Miao, S. Bai, B. Li, Z. Zheng, L. Zhu, Z. Shen, Y. Xi, P. Zhang, D. Cai, M. Zhang, and M. Xu. Flow event telemetry on programmable data plane. In *Proceedings of the 2020 ACM SIGCOMM Conference (SIGCOMM'20)*, Virtual Event, Aug. 2020.

[120] H. Zhu, Z. Bai, J. Li, E. Michael, D. R. K. Ports, I. Stoica, and X. Jin. Harmonia: Near-linear scalability for replicated storage with in-network conflict detection. In *Proceedings of the 2019 International Conference on Very Large Data Bases (VLDB'19)*, Los Angeles, CA, Nov. 2019.

Dynamic Scheduling of Approximate Telemetry Queries

Chris Misa
University of Oregon

Walt O'Connor
University of Oregon

Ramakrishnan Durairajan
University of Oregon

Reza Rejaie
University of Oregon

Walter Willinger
NIKSUN, Inc.

Abstract

Network telemetry systems provide critical visibility into the state of networks. While significant progress has been made by leveraging programmable switch hardware to scale these systems to high and time-varying traffic workloads, less attention has been paid towards efficiently utilizing limited hardware resources in the face of dynamics such as the composition of traffic as well as the number and types of queries running at a given point in time. Both these dynamics have implications on resource requirements and query accuracy.

In this paper, we argue that this dynamics problem motivates reframing telemetry systems as *resource schedulers*—a significant departure from state-of-the-art. More concretely, rather than statically partition queries across hardware and software platforms, telemetry systems ought to decide on their own and at runtime *when* and for *how long* to execute the set of active queries on the data plane. To this end, we propose an efficient approximation and scheduling algorithm that exposes accuracy and latency tradeoffs with respect to query execution to reduce hardware resource usage. We evaluate our algorithm by building *DynATOS*, a hardware prototype built around a reconfigurable approach to ASIC programming. We show that our approach is more robust than state-of-the-art methods to traffic dynamics and can execute dynamic workloads comprised of multiple concurrent and sequential queries of varied complexities on a single switch while meeting per-query accuracy and latency goals.

1 Introduction

Network telemetry systems provide users (e.g., network operators, researchers) with critical insights into the state of the network by collecting information about individual packets and processing this information into high-level features in near real-time. Typically, these features are the results of user-defined queries, where a query is expressed as a sequence of high-level operations such as filter and reduce [22, 33, 43]. Generated query results drive management decisions such as deploying defensive measures in the face of an attack or updating routing to avoid congestion. A key functionality of telemetry systems is to determine how best to leverage available resources (*e.g.*, network hardware resources, such as switch ASICs or NICs; software-programmable resources, such as general-purpose CPUs) to execute a given set of queries. Due to massive traffic volumes and often stringent timing requirements, state-of-the-art telemetry systems typically make use of programmable network hardware (*e.g.*, programmable switch ASICs [2, 4, 5]) and also apply approximation techniques (*e.g.*, sketches [24, 38, 39]).

In executing user-defined queries, telemetry systems must cope with two independent and challenging sources of dynamics. First, the resources required to execute any given query depend on the underlying distributions (*i.e.*, composition) of network traffic. For example, a DDoS-detection query that counts the number of sources contacting each destination might require a counter for each destination active on the network, but the number of active destinations may vary over time [38]. The accuracy guarantees of state-of-the-art approximation techniques like sketches [39] likewise depend on traffic distributions so that if these distributions change, accuracy can no longer be guaranteed. Second, the number and type of concurrent queries submitted by a user can vary over the system's deployment. For example, an operator might need to submit followup queries to pinpoint the root cause of increased congestion. Both these sources of dynamics affect data plane resource usage implying that telemetry systems must dynamically adjust resource allocations.

Several recent efforts [38, 43] have made progress towards coping with both of these sources of dynamics individually and in isolation, but do not address challenges arising from their simultaneous presence in network telemetry systems. For example, ElasticSketch [38] presents a method for dynamically coping with changes in traffic rate and distribution. However, this effort relies on a fixed flow key which forces users to reload the switch pipeline to change queries. On the other hand, Newton [43] describes a technique to update query operations during runtime which enables users to dynamically add and remove queries as their monitoring needs

change. However, Newton does not consider the problem of adjusting resource allocations between concurrent queries as traffic composition changes. To the best of our knowledge, no recent work addresses these simultaneous sources of dynamics in an efficient switch hardware based system.

In this work, we argue that, in order to simultaneously address these sources of dynamics, *telemetry systems should be reframed as active resource schedulers for query operations*. In particular, telemetry systems must manage finite switch hardware processing resources while adapting to varying numbers and types of queries as well as varying traffic composition. To support this argument, we make the following key contributions.

Time-division approximation method. Viewing telemetry systems as online schedulers enables a new approximation technique based on time-division approximation. At a high-level, this technique observes that query operations do not need to run all the time. Instead, operations can execute during strategically placed sub-windows of the overall time window (*e.g.*, an operation could execute for 3 of 8 equal-duration sub-windows of a 5 s overall time window). This technique is grounded in cluster sampling theory which allows us to estimate error and future resource requirements.

Adaptive scheduling algorithm. We provide a closed loop adaptive scheduling algorithm which leverages time-division approximation to execute operations from many user-defined queries on a single switch ASIC. Our scheduling algorithm leverages multi-objective optimization to balance between multiple high-level goals such as prioritizing accuracy, latency, or reduced volume of reported data across queries.

Evaluation in a functional hardware prototype. To evaluate our proposed techniques, we implement *DynATOS*,[1] a telemetry operation scheduling system which leverages programmable switch hardware to answer dynamically submitted queries. Our current implementation of *DynATOS* assumes a single runtime programmable switch hardware capable of executing a restricted number of primitive operations as supported by a telemetry module found in a widely available off-the-shelf switch ASIC. We evaluate *DynATOS* on our hardware prototype and through simulation showing that (i) time-division approximation is more robust than sketches to changes in traffic dynamics while offering a similar accuracy, overhead tradeoff space, (ii) our adaptive scheduler is able to meet query accuracy and latency goals in the presence of traffic and query dynamics, and (iii) the overheads in our scheduling loop are minimal and dominated by the time required to report and process intermediate results from the switch—an overhead which can be mitigated significantly by leveraging fully programmable switch hardware.

2 Background & Motivation

2.1 Dynamic Telemetry Use Cases

Example 2.1. Consider a scenario where a telemetry system is executing the DDoS and port scanning detection tasks described in Sonata [22][2]. The first stage of these tasks finds a set of distinct elements in each time window or epoch (*e.g.*, IPv4 source, destination pairs every epoch for DDoS). Suppose traffic follows a stable pattern for several epochs with only small changes in the number of distinct elements considered by both tasks and that the telemetry system adjusts resource allocations for these two queries to achieve good accuracy. Now, suppose at some later epoch traffic changes so that a much larger number of sources are seen (either due to a natural event like a flash crowd or due to an actual DDoS attack). This larger number of sources increases the number of pairs that both queries must keep track of and either more resources will need to be allocated or accuracy will suffer.

While this example only considered a pair of queries, in realistic settings operators likely need to monitor for a wide variety of attacks simultaneously (*e.g.*, the 11 queries described in Sonata [22]). Moreover, features like number of sources or destinations commonly overlap in these types of attack detection queries so that an anomalous change in one feature may upset the resource requirements of a large number of simultaneous queries.

Example 2.2. Consider a scenario where a network operator wants to understand the root cause of TCP latency on their network. In this scenario, the operator would like to first run queries to detect when latency increases and for which hosts or subnets [18]. Once detected, the operator must submit a large number of queries to test possible causes of high latency such as re-transmissions or deep queues [33] with filter operations so that these queries only apply to the flows experiencing latency. Note that the debugging phase may require several rounds of querying with tens of simultaneous queries in each round before the root cause of the latency can be determined.

While the above examples focus on two particular tasks, the underlying concepts—of dealing with large shifts in query resource requirements caused by changes in traffic and of executing multiple queries over time in a dependent manner— are commonly encountered in network operations.

2.2 Ideal Telemetry System Requirements

In light of the above-mentioned examples, an ideal telemetry system should support the following requirements.
R1: Query diversity. Marple [33] and Sonata [22] outline how a small set of parameterized stream processing operators can enable a wide range of telemetry queries. Telemetry systems must support these kinds of generic query description interfaces, allowing filtering over packet header values,

---

[1]*DynATOS* stands for Dynamic Approximate Telemetry Operation Scheduler.

[2]The DDoS task finds destinations receiving from large numbers of distinct sources and the port scanning task finds sources sending to a large number of distinct destination ports.

Approach	R1	R2	R3	R4	R5
Static switch-based	✓				✓
Runtime-programmable	✓	✓		✓	✓
Dynamic allocation		✓	✓	✓	✓
Sketch-based	✓	✓			✓
Software-based	✓	✓	✓	✓	
DynATOS	✓	✓	✓	✓	✓

Table 1: Summary of how different approaches relate to the requirements of § 2.2.

grouping by arbitrary header fields, chaining operations, and joining the results of multiple operation chains.

R2: Approximate execution. Executing telemetry queries over the massive volumes of data flowing through networks poses heavy resource requirements. Furthermore, many telemetry queries are equally effective when computed approximately [30]. Therefore, telemetry systems should expose approximation techniques that allow trading off reduced result accuracy for lower resource requirements.

R3: Traffic dynamics. Composition of traffic changes over time, and changes may be slow, regular, and easy to predict (*e.g.*, daily cycles) or fast and hard to predict (*e.g.*, flash crowds). As discussed in Example 2.1, these changes in traffic composition lead to changes in the resource requirements for different groups of queries. Telemetry systems should robustly handle these changes without compromising query accuracy or latency [38].

R4: Query dynamics. The queries a network operator needs to run change over time. Some of these changes may be infrequent (*e.g.*, adding new queries to monitor a newly deployed service), while some of these changes may be rapid and time-sensitive (*e.g.*, adding new queries to debug a performance anomaly or to pinpoint and block a network attack). Telemetry systems should be able to handle these dynamic query arrivals and removals, realizing updates within a few milliseconds and without incurring network downtime [43].

R5: Switch hardware acceleration. Due to massive traffic volumes, stringent timing requirements, and the limited speed of a single CPU core, executing telemetry queries on CPU-based systems is prohibitively expensive [22]. As a result, telemetry systems must leverage resource-constrained hardware targets [2, 4, 5] to accelerate query execution.

2.3 State-of-the-art and their Limitations

State-of-the-art approaches each satisfy a subset of the requirements set forth above, but face limitations which hinder their ability to satisfy all requirements simultaneously.

Static switch-based approaches. Marple [33] and Sonata [22] compile traffic queries into static hardware description languages like P4 [10], demonstrating the efficiency of switch hardware in computing query results. However, these approaches fail to satisfy R4 since changing queries incurs seconds of network downtime (see [43]).

Runtime-programmable approaches. Recently, Beau-Coup [14] and Newton [43] demonstrate techniques to allow network operators to add and remove queries at runtime without incurring downtime. These efforts lay a technical foundation to address R4, but do not address the challenge of R3.

Dynamic allocation approaches. DREAM [30] and SCREAM [31] develop dynamic allocation systems for telemetry operations addressing both R3 and R4. However, these approaches do not satisfy R1 because they require query-specific accuracy estimators.

Sketch-based approaches. Many telemetry efforts address R2 by leveraging sketches [15, 16, 28, 39, 42] to gather approximate query results under the stringent operation and memory limitations faced in the data plane. However, the accuracy of sketches is tightly coupled to both the resources allocated (*e.g.*, number of hash functions or number of counters) and the underlying composition of traffic (*e.g.*, number of flows) making sketches insufficient for R3 and R4. An exception to this is ElasticSketch [38] which addresses R3 head on by dynamically adapting to varying traffic compositions. However, ElasticSketch fails to address R4 or R1 since flow keys are fixed in the sketch's implementation.

Software-based approaches. Several prior efforts leverage the capabilities of general-purpose CPUs to process traffic queries. For example, Trumpet [32] installs triggers on end hosts, OmniMon [25] and switch pointer [37] share tables between end hosts and switches in network, and SketchVisor [23] and NitroSketch [27] tune sketch-based approximation techniques for virtual switches. While these approaches work well in settings like data centers where all infrastructure is under a single administrative domain, in many settings (*e.g.*, campus or enterprise networks) it is too expensive (in terms of infrastructure cost and/or latency) to pass all packets through software and impractical to instrument end hosts.

Scheduling distributed stream processing operations. Several efforts [26, 34–36, 41] address the challenge of efficiently scheduling stream processing operations to maximize resource utilization. However, these efforts do not consider the particular types of accuracy and latency constraints encountered in scheduling telemetry operations on switch hardware.

Limitations of current hardware-based approaches. To illustrate the limitations of current static approaches [22, 33, 43] in dealing with R3 and R4, we implement the two queries mentioned in Example 2.1 and run them over a traffic excerpt from the MAWILab [17] data set which features pronounced traffic dynamics. This excerpt starts with relatively stable traffic, then suddenly, due to an actual DDoS attack or other causes (which we do not claim to identify), around the 20th 5 s time window (or *epoch*) contains a large number of sources sending regular pulses of traffic. As suggested in [22, 43], we use bloom filters tuned for the initial normal traffic to approximate the lists of distinct pairs required by the first stage of both queries.
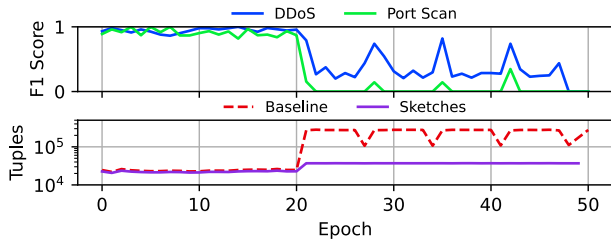
Figure 1: Accuracy of concurrent DDoS and port scanning queries under extreme traffic dynamics.

Figure 1 shows the F1 score[3] of these approximate query implementations along with the number of tuples returned to the collector in each epoch. Before the change in number of sources, the approximation methods for both queries return highly accurate results while sending relatively few tuples. However, when the number of sources increases, the approximation accuracy of both queries suffers since the actual number of ground truth tuples (the "Baseline" trace) far exceeds the number each query was tuned for. Taking the static approach in this example shows that *when certain events of interest occur, the accuracy of multiple queries can be significantly impacted due to fixed assumptions about traffic composition.* Of course, the telemetry system initially could have tuned these queries for the anticipated number of sources, but this would lead to significant wastage of resources under normal traffic conditions and it is hard to know what to tune for without prior knowledge of the anomaly.

2.4 Design Challenges

To elucidate why prior efforts fail to meet the requirements put forth in § 2.2, we next describe the following high-level design challenges.

D1: Approximating generic query results. Efforts like Marple and Sonata develop expressive query description languages which map into data plane computation models. However, approximation of query operations is often necessary due to limited data plane resources and massive traffic volumes. It is unclear how state-of-the-art approximation methods can be leveraged to work with queries expressed in languages like Marple or Sonata. As illustrated in § 2.3, the currently proposed baseline approach of simply replacing stateful reductions in Sonata queries with sketch-based primitives requires prior knowledge of worse-case traffic situations and does not perform well under dynamic traffic scenarios.

D2: Estimating accuracy of approximations. Approximate query results must be accompanied with a sound estimate of their accuracy. This is critical for operators to understand the system's confidence in detecting a particular event or reporting a particular metric and equally critical in dynamic telemetry systems to inform the balance of resources between approximate queries. Prior efforts have made progress towards

this goal [24, 30, 31], but none anticipate accuracy estimation for current state-of-the-art generic query descriptions.

D3: Allocating finite hardware resources among variable sets of queries under traffic dynamics. Very few prior efforts address the need of a telemetry system to evaluate multiple concurrent queries on finite hardware resources. In order to handle traffic dynamics, such a system must dynamically update resource allocations based on the estimated accuracy of each query. Moreover, since it is possible that the given resources will be insufficient to meet the accuracy of all queries, such a system must enable operators to express query priorities and allocate resources with respect to these priorities.

3 *DynATOS* System Design

3.1 Overview

To tackle the above-mentioned challenges, we build *DynATOS*. At its core, *DynATOS* is composed of three main components as shown in Figure 2. Network operators submit queries to the scheduler through a high-level REST API which performs initial query validation and returns a status message along with a description of the expected query result format. The scheduler then translates queries into their primitive operations and constructs schedules for how these operations should be run on switch hardware. These schedules are then handed to a runtime component which communicates with switch hardware to execute the primitive operations and collect intermediate results. Once ready, the runtime component gathers all results and passes them back to the scheduler and operators.
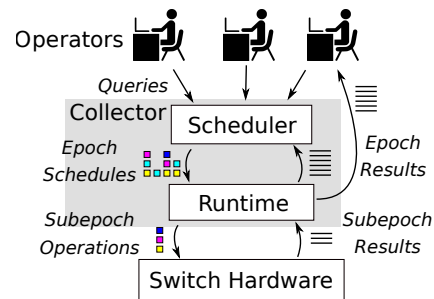
Figure 2: Architecture of *DynATOS*.

3.2 Preliminaries

Scheduling horizon. Since queries can arrive at any time, we must decide when and for how far into the future resources should be scheduled. We first examine several possible approaches to this problem, then describe our approach in the next paragraph. One option is to compute the schedule each time a new query arrives and adjust all existing queries to the new schedule. While this option minimizes the time a query has to wait before it can start executing, it complicates the realization of accuracy and latency goals since the duration of the scheduling horizon (*i.e.*, until the next query arrives) is unknown when forming the schedule. Alternatively, we could compute the new schedule each time all queries in the prior

---

[3]Computed by comparing with ground truth, the F1 score is a measure of query accuracy defined as the harmonic mean of precision and recall.

schedule terminate. While this option ensures schedules can be executed exactly as planned, newly submitted queries may experience a longer delay.

We choose, instead, to make scheduling decisions at fixed windows of time which we call *epochs* (*e.g.*, every 5 s). This allows a balance between the two schemes mentioned above: queries must wait at most the duration of one epoch before executing and during an epoch queries are ensured to execute according to the schedule. In particular, we divide the scheduling epoch into N subepochs and our scheduler assigns subsets of the submitted queries to each subepoch as shown in Figure 3. Subepochs provide flexibility to schedule different queries at different times while also providing concrete resource allocation units. Queries submitted during an epoch are checked for feasibility and only considered in the following epoch. For example, in the figure, Q4 is added sometime during epoch 2, but cannot be scheduled until epoch 3. During the epoch, the scheduler collects intermediate results for each subepoch in which a query is executed and aggregates these subepoch results based on the query's aggregation operation. Once an epoch completes, results of complete queries are returned, while new and incomplete queries are considered for the next epoch. For example, in Figure 3 Q3 completes execution in the second subepoch of epoch 2 and its results are returned during the scheduler invocation before epoch 3. We further assume that each query executes over traffic in a single epoch and telemetry tasks requiring longer measurement durations than our scheduling epoch can simply re-submit queries.
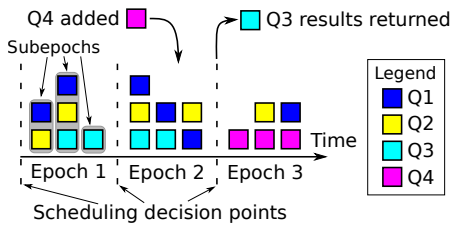
Figure 3: Example of scheduling 4 queries with $N = 3$ subepochs per epoch.

3.3 Key Ideas

We develop a novel approximation method to address the challenge of gathering approximate results for generic queries (D1). In particular, our method leverages cluster sampling theory to estimate the results of the first aggregation operator in multistage queries. For example, in the DDoS query we only approximate computation of the distinct source, destination pairs list and execute all subsequent operations exactly. The intuition behind this is that each operator in a telemetry query tends to reduce the volume of data passed to the next operator. Therefore, reducing the resource requirements and volume of data emitted from the first aggregation reduces the load on all subsequent operators.

§ 4 describes how our approximation method can provide

sound estimates of result accuracy without prior assumptions about traffic characteristics (addressing D2). Note that the accuracy estimates used in many sketch methods are dependent on traffic characteristics (which can be estimated by auxiliary queries or offline analysis) [39]. Our method, on the other hand, uses cluster sampling to estimate result accuracy based on observations from a single epoch independently of traffic characteristics. Moreover, by leveraging observations of feature variance in prior epochs, we can predict resource requirements for a desired accuracy level in future epochs. This feedback loop allows our system to dynamically adjust per-query allocations as traffic distributions change.

To address D3, we integrate our approximation technique in a scheduler that determines how a number of concurrent queries should be executed on a single switch hardware, balancing resources between queries to satisfy accuracy and latency goals set by operators. As described in § 5, our scheduler uses a novel multi-objective optimization formulation of the problem of when to run which queries given query priorities and resource constraints. This formulation allows the scheduler to balance between the goals of multiple concurrent queries, sometimes allocating less than the exact number of subepochs when queries have lower priority and resources are scarce (*e.g.*, due to a large number of concurrent queries).

Finally, we develop a runtime system leveraging these ideas to efficiently execute schedules on switch hardware, gather intermediate results, apply factors to correct for sampling, and return results to network operators in a high-level format. Operators can then decide to execute new queries in the subsequent epoch, or to re-execute the current queries based on these results.

3.4 Limitations and Assumptions

Monitoring problems addressed by *DynATOS*. The types of traffic features which can be monitored by queries in *DynATOS* are subject to the following assumptions.

- Feature values do not fluctuate excessively over measurement durations of one or two seconds.
- The monitoring task can be implemented using features gathered at a single point in the network.
- Features are constructed from packet header fields and/or other switch-parsable regions of the packet.
- Features can be computed using atomic filter, map, and reduce operations.

Under these assumptions monitoring tasks like detecting microbursts [13], identifying global icebergs [19], and detecting patterns in TCP payloads [9] cannot be efficiently executed using *DynATOS*. However, as evidenced by the body of prior efforts with similar assumptions (*e.g.*, [22, 30, 33]) and the concrete examples discussed in § 2.1, *DynATOS* can still be used for a wide variety of useful tasks.

Switch hardware model. In the following, we assume a restricted runtime programmable switch hardware model. In this model, switch hardware is able to execute a fixed set

of Sonata [22] operators, in particular, a filter operator followed by a reduce operator. However, similar to Newton [43], our switch hardware allows arbitrary parameterization of these operators *at runtime*. For example, switch hardware could execute the filter and reduce commands required by the Sonata TCP new connections queries for a period of time, then quickly (*e.g.*, within a few milliseconds) be re-programmed to execute the filter and reduce commands required by the Sonata DDoS query. We note that our scheduling methods are independent of this particular switch hardware model and could readily be applied to more fully programmable ASICs [5,10].

Network-wide scheduling. Ultimately, operators need to query traffic across different logical or physical domains of their network. This implies that telemetry systems should collect information from a distributed set of switches (or other monitoring points) and provide a global view of network traffic. In this work, we consider only a single monitoring point (*e.g.*, a critical border switch) and leave the challenges of distributed scheduling of telemetry operations to future work. Nonetheless, a single switch deployment on a enterprise or data center border switch can still be highly effective in executing the types of queries considered.

4   Time-Division Approximation in *DynATOS*

Accuracy tradeoff. Given fixed scheduling epochs, we can trade off accuracy for reduced resource requirements by sampling a subset of the subepochs in which to execute a particular query. We leverage cluster sampling theory [29] to expose this tradeoff while maintaining accuracy goals. Cluster sampling is a good fit for situations like dynamically scheduled query operations where the cost of sampling large groups of the population (*i.e.*, subepochs) is significantly lower than the cost of sampling individual population members (*i.e.*, packets) [29]. In particular, we assume sending the aggregate results (computed in switch hardware) of each sampled subepoch to software is cheaper than sending individual sampled packets to software.

Consider the case where a particular query executes in n of the N total subepochs and let $t_{i,j}$ be the query's result in the i-th subepoch of the j-th epoch, n_j be the number of subepochs in which the query executed in the j-th epoch, E be the total number of epochs in which the query is executed, and $s_{t_j}^2$ be the sample variance of the $t_{i,j}$'s in the j-th epoch. We use the unbiased estimator,[4]

$$\hat{t}_E = \frac{1}{E}\sum_{j=1}^{E}\hat{t}_j = \frac{1}{E}\sum_{j=1}^{E}\frac{N}{n_j}\sum_{i\in S_j}t_{i,j} \qquad (1)$$

which has standard error

$$SE(\hat{t}_E) = \frac{N}{E}\sqrt{\sum_{j=1}^{E}\left(1-\frac{n_j}{N}\right)\frac{s_{t_j}^2}{n_j}} \qquad (2)$$

---

[4]See § A for a full discussion of the derivation of these equations from cluster sampling theory.

to estimate query results and determine when accuracy goals have been fulfilled. We rearrange Equation 2 as

$$n^{acc} = \frac{s_{t_E}^2 N^2}{E^2\sigma^2 - \left(\sum_{j=1}^{E-1}\mathrm{Var}\left(\hat{t}_j\right)\right) + Ns_{t_E}^2} \qquad (3)$$

to estimate the number of subepochs in which a query should execute in the E-th epoch to fulfill a given standard error target σ assuming the query has already executed in the previous $E-1$ epochs without fulfilling σ. Note that if $\sigma = 0$, then $n^{acc} = N$ and the query will be executed in all of the subepochs in its first epoch. As σ increases, n^{acc} decreases freeing more of the subepochs for other queries.

Latency tradeoff. In addition to the accuracy tradeoff discussed above, we can tradeoff result latency for reduced resource requirements by executing a query's operations across several epochs. The key observation enabling this tradeoff is that by spreading the sampled subepochs over several epochs, the query can reduce its per-epoch requirements while still attaining its accuracy goal. Operators leverage this tradeoff by specifying larger latency goals on queries that do not require fast returns.

Suppose a particular query has a latency goal of $\tilde{E}$ epochs. We need to estimate the number of subepochs in which the query should be allocated n^{lat} in the e-th epoch with $1 \le e \le \tilde{E}$. First, we break the sum in Equation 2 into *past* ($1 \le j < e$) and *future* ($e < j \le \tilde{E}$) components. We then have,

$$n^{lat} = \frac{s_{t_E}^2 N^2}{\tilde{E}^2\sigma^2 - N^2\left(past + future\right) + Ns_{t_E}^2} \qquad (4)$$

While the *past* component can be calculated directly using observations from prior epochs, the *future* component must be estimated based on the number of subepochs the query expects to receive in future epochs. Operators can tune this expected number of subepochs based on current and expected query workloads.

Correcting distinct operators. While the previous sections discuss foundations for making sound approximations of packet/byte counts, many useful queries also involve identifying and counting distinct elements. We leverage the Chao estimator without replacement[5] to correct estimates of a common class of distinct count queries such as the DDoS query considered in § 2.1. Similar to the cluster sampling estimators described in this section, the Chao estimator can be used to obtain point and standard error estimates based only on the observed sample.

5   Scheduling in *DynATOS*

5.1   Optimization Formulation

We cast the task of generating query schedules as an optimization problem and adapt well-known techniques to generate

---

[5]See § A.3 for details.

schedules through this casting. While this section details our casting of the problem, § 5.2 describes the challenges inherent in applying optimization techniques in a real-time setting such as ours.

We apply our optimization formulation every epoch to determine which queries should execute in each of the N subepochs as shown in Algorithm 1. First, in line 2 we use the DISENTANGLE method of Yuan et al. [40] to break the submitted queries Q into disjoint traffic slices K and save the mapping between queries and slices in $s_{i,k}$. Line 3 then computes the minimum number of stateful update operations required by the reduce operators of all queries in each particular slice. These steps are necessary given our single-stage switch hardware model (§ 3.4). Next, lines 4 through 6 compute estimates of the memory and subepoch requirements of each query. Finally line 7 creates and solves the optimization problem described below. If a feasible solution cannot be found, line 9 falls back to a heuristic scheduling method described in § 5.2.

Algorithm 1 Method for determining subepoch schedule

1: **procedure** GET-SCHEDULE(Q, u, SE)
2:     $K, s \leftarrow$ DISENTANGLE(Q)
3:     $U \leftarrow$ COMBINE-UPDATES(u, K, s)
4:     $m \leftarrow$ ESTIMATE-MEMORY
5:     $n^{acc} \leftarrow$ EQUATION 3(σ)
6:     $n^{lat} \leftarrow$ EQUATION 4(σ, E)
7:     $d \leftarrow$ SOLVE-OPTIMIZATION
8:     **if** d is infeasible **then**
9:         $d \leftarrow$ GET-HEURISTIC-SCHEDULE
10:     **end if**
11: **end procedure**

Inputs. Table 2 shows the particular inputs and outputs of this optimization problem. Of the input variables, t_k, u_i, $s_{i,k}$, T, A, and M are known exactly based on submitted query requirements and available switch resources, while m_i, n_i^{acc}, and n_i^{lat} must be estimated based on observation of past epochs. Our current implementation uses EWMA to estimate m_i and $s_{t_E}^2$ (as required by n_i^{acc} and n_i^{lat}) independently for all update operation types. We leave exploration of more sophisticated estimation approaches to future work. Scheduling decisions are encoded in the $d_{i,j}$ indicator variables which determine which queries should execute in each subepoch. We do not consider the division of switch memory between queries since memory is dynamically allocated during the aggregation operation (see § 3.4).

Constraints. We impose the constraints shown in Table 3 to satisfy two high-level requirements: (i) respecting switch resource limits (C1, C2, C3) and (ii) forcing minimal progress in each query and ensuring variance estimates are well-defined (C4). Note that C2 captures the fact that if two queries rely on the same update operation, they can be merged to use a single ALU. In the case that the estimated quantity m_i turns out to

Variable	Description
Q	index set of queries ready for execution
SE	index set of subepochs
K	index set of all disjoint traffic slices
U_k	index set of all update operations in slice k
t_k	number of TCAM entries required by slice k
u_i	index of update operation required by query i
$s_{i,k}$	indicator that query i requires slice k
m_i	memory required in each subepoch by query i
n_i^{acc}	number of subepochs required for accuracy goal for query i (§ 4)
n_i^{lat}	number of subepochs required for latency goal for query i (§ A.2)
T	total available TCAM entries
A	total number of available switch ALUs
M	total available SRAM counters
$d_{i,j}$	indicator that query i executes in subepoch j

Table 2: Variables used in optimization formulation of scheduling problem. The sole outputs $d_{i,j}$ determine the schedule for the next epoch.

C1: $\forall j \in SE, \ \sum_{k \in K} t_k \mathbf{I} \left[\bigvee_{i \in Q} d_{i,j} s_{i,k} = 1 \right] \leq T$

C2: $\forall j \in SE, \ k \in K, \ \sum_{u \in U_k} \mathbf{I} \left[\bigvee_{i \in Q} d_{i,j} s_{i,k} \mathbf{I}[u_i = u] = 1 \right] \leq A$

C3: $\forall j \in SE, \ \sum_{i \in Q} d_{i,j} m_i \leq M$

C4: $\forall i \in Q, \ \sum_{j \in SE} d_{i,j} \geq 2$

Table 3: Scheduling problem constraints to respect (C1) TCAM capacity requirement, (C2) switch ALU capacity, (C3) SRAM capacity, and (C4) query minimal progress requirement. $\mathbf{I}[]$ is the indicator function.

O1: $\text{minimize} \sum_{i \in Q} \left| \left(\sum_{j \in SE} d_{i,j} \right) - n_i^{acc} \right|$

O2: $\text{minimize} \sum_{i \in Q} \left| \left(\sum_{j \in SE} d_{i,j} \right) - n_i^{lat} \right|$

O3: $\text{minimize} \sum_{i \in Q, j \in SE} d_{i,j} m_i$

Table 4: Objective functions considered in the multi-objective formulation.

be violated by traffic conditions in the subsequent epoch, we simply drop new aggregation groups once the available switch memory is totally consumed.

Objectives. In computing the schedule of each epoch, we consider the objective functions listed in Table 4. O1 seeks to satisfy accuracy goals by minimizing the distance to the value of n^{acc} computed in Equation 3, O2 seeks to satisfy latency goals by minimizing the distance to the value of n^{lat} computed in Equation 4, and O3 seeks to limit the maximum volume of data that needs to be returned from the switch in a single subepoch. We expose the Pareto front of these objective functions using linear scalarization which allows operators to express the importance of each objective by submitting weights and is computationally efficient.

5.2 Challenges of Online Optimization

Unlike prior work (*e.g.*, [22]), the inputs to our optimization problem are dependent on task dynamics (*e.g.*, the set Q can vary each epoch) and traffic dynamics (*e.g.*, the suggested n_i^{acc} could increase in response to increased traffic variability). Hence, we must solve the optimization problem independently in each epoch. However, invoking an optimization solver in an online scheduling method is fraught with challenges. First, certain combinations of inputs and constraints can lead to infeasible problems where it is impossible to satisfy all constraints. Second, since integer programming is a well known NP-complete problem, finding an optimal solution can take exponential time in the worst case. In what follows, we describe several precautions that we take in the design of our scheduler to ensure these challenges do not adversely affect the performance of the telemetry system.

Dealing with infeasible queries. Our first strategy to deal with infeasible optimization problems is to require that all submitted queries can be executed on the given switch resources in the absence of other queries. In particular, if a query requires more than T TCAM entries, A ALUs, or M counters, the scheduler must reject that query outright, since it will not be able to execute on the given switch hardware. This ensures that our scheduler can always make progress on the current pool of submitted queries by selecting a single query and allocating the full switch resources for all subepochs. We note that a query partition scheme similar to Sonata [22] could be added to our system to handle this case more elegantly, but leave this to future work.

Dealing with slow optimizations. To deal with the potentially exponential time that could be required to converge to an optimal solution, we limit the duration of time spent in the optimization algorithm to an acceptable fraction of total epoch time. This method, known as early stopping, is a well-known technique to gather feasible, good, if not fully optimal solutions. When the optimization process stops due to this time limit, the current solution must still be checked for feasibility and only allowed to execute if it is, in fact, feasible.

Fail-safe. In cases where the optimization problem is either proven infeasible or times out before converging, we fall back to a simple heuristic "fail-safe" mode of scheduling. We also deny all new query submissions when in fail-safe mode to notify the operator that the system is currently saturated and to prevent the accumulation of a large backlog which could cause the optimization problem to remain infeasible in future epochs. Our simple heuristic fail-safe scheduling scheme greedily selects the query closest to its deadline and allocates this query fully to switch resources. To increase progress in fail-safe mode, we also add other queries that use the same or a subset of the selected query's traffic slices until either the memory or ALU limit is reached. Since queries scheduled in this mode execute for each subepoch, $n_j/N = 0$ for that epoch ensuring progress towards accuracy targets, though some queries may suffer increased latency.

Another approach to dealing with situations where a feasible schedule cannot be found is to send slices of traffic to the collector and compute query results in software. In this approach queries running during fail-safe mode could still meet tight latency goals at the expense of increased load on the collector. Depending on the nature of situation triggering fail-safe mode, this could impose infeasible processing loads on the collector or lead to excessive congestion between switch and collector. In future work, we plan to investigate solutions to this problem including combinations of heuristic scheduling and moving query operations to software.

6 Evaluation

In this section, we describe our evaluation of *DynATOS* and demonstrate the following key results.

- The time-division approximation technique in *DynATOS* is more robust than state-of-the-art in the face of traffic dynamics and offers comparable performance to state-of-the-art sketch-based approximate techniques (§ 6.2).
- The scheduling method in *DynATOS* handles dynamic query workloads with up to one query every second and leverages specific accuracy and latency goals to reduce per-query resource usage (§ 6.3).
- Latency overheads in *DynATOS* are minimal and dependent on the load on the collector and the number of queries which must be updated in switch hardware (§ 6.4).

6.1 Experimental Setup

Setting. We evaluate *DynATOS* on a BCM 56470 series [8] System Verification Kit (SVK) switch running BroadScan [1] which implements the telemetry operations described in § 3.4. Our version of BroadScan has $A = 8$ parallel ALU operators, and a flow table with $M \approx 9$MB of memory. A software agent on the switch's CPU manages reconfiguration of hardware in response to requests from the collector. Our collector and scheduling software runs on a server with an Intel Xeon Gold 5218 CPU at 2.3Ghz and 383GB memory. This server is equipped with a 40Gb Mellanox MT27700-family network card connected directly to the SVK's data plane. A separate 10Gb Intel X550T network card on the same server connects to the SVK's management interface to manage updates to hardware configuration as schedules execute.

Traces. Unless otherwise stated, we replay a trace from the MAWILab traffic data set (Sept. 1st, 2019) [17] using `tcpreplay` [7]. We selected this trace as a baseline because some of its features are static while others are more dynamic.

Default parameters. We use five-second scheduling epochs to allow sufficient measurement duration without incurring excessive delay of results which must wait for epoch boundaries. We divide epochs into $N = 8$ subepochs so that the schedule has sufficient options for arranging queries without making subepochs too short to generate useful samples. We set objective weights to balance between priorities and suppose queries will get all future subepochs when evaluating

Equation 4. Queries are submitted with realistic values of σ based on baseline measurements of their variances in the trace. We set $\alpha = 1/2$ in the EWMA estimation described in § 5.1. Bars show median and error bars show 5th and 95th percentiles over all epochs of the trace.

Query workloads. We use *DynATOS* to implement four of the telemetry queries originally introduced by Sonata [22] and used in several recent efforts. Our hardware model handles a fixed sequence of filter and reduction operations so we implement the remaining query operations in software. This scenario is equivalent to Sonata with a limited number of switch hardware stages. We report the accuracy of approximate implementations of these queries as F1 score (the harmonic mean of precision and recall) by comparing against ground truth computed offline. In addition to static queries, we generate dynamic query workloads based on random processes to evaluate *DynATOS* (see § 6.3). To the best of our knowledge, there is no comparable publicly-available dynamic query workload benchmark. Our workloads are publicly released at [6] to support validation of our results and to facilitate benchmarking of similar systems in the future.

Implementation. We implement the *DynATOS* scheduler in ~14k lines of C and C++. Following ProgME [40], we use BDDs to represent query filter conditions in our implementation of the DISENTANGLE algorithm (§ 5.1). We use the open source CBC implementation [3] to solve the optimization problems described in § 5.1. Our implementation also defers some result processing operations to the time spent waiting for results from switch hardware to improve efficiency.

Comparisons. We compare *DynATOS* with ElasticSketch [38], Newton [43], and SketchLearn [24]. We modified the implementations of both ElasticSketch and SketchLearn to support the filter and reduce operations required by several of the Sonata [22] queries. Though we were unable to locate a publicly available implementation of Newton, we implemented its sketch-based approach to approximating Sonata's primitive operators. In particular, we use count-min sketch [15] to approximate the `reduce` operator and a bloom filter [20] to approximate the `distinct` operator.

6.2 Performance of Time-Division Approximation

Robustness in the face of traffic dynamics. To address the question of what happens when traffic composition changes significantly we consider an excerpt from the MAWILab dataset taken on Nov. 14th, 2015. As shown in Figure 4, this excerpt features nominally static traffic followed by a dramatic surge in the number of sources around 100 seconds into the trace.

To understand how different methods handle this change in traffic dynamics, we first tune each method's parameters to achieve high accuracy (F1 > 0.9) on the first 100 seconds of the excerpt, then run the method with these parameters over the entire excerpt. Since it is possible that this anomaly
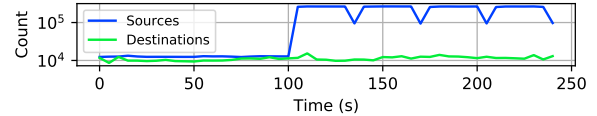
Figure 4: Number of distinct sources and destinations in excerpt from MAWILab data on Nov. 14th, 2015.

was cause by some form of DDoS attack, we run the DDoS query in this scenario to locate the victim of the attack. This is intended to reflect a realistic situation where a method was deployed and tuned for a particular traffic composition, which then changes. In real deployments, such changes could be caused by attacks or performance anomalies and represent the moments when data collected from a telemetry system is most critical.
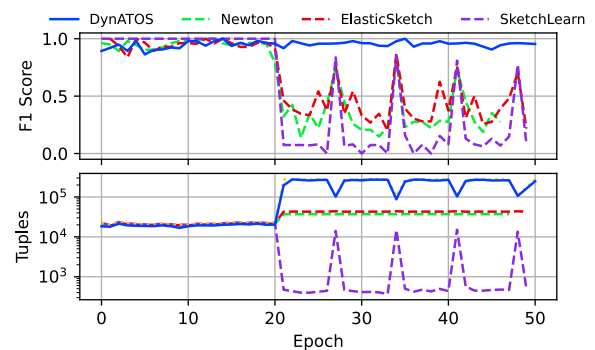
Figure 5: Performance of different methods on the 2015 MAWILab excerpt shown in Figure 4.

Figure 5 shows the F1 score and number of tuples returned to the collector in each epoch over the trace excerpt. All methods achieve high accuracy for the first 20 epochs, but then when the number of sources increases after the 20th epoch, they diverge significantly. First, we note that *DynATOS* is able to maintain high accuracy where other methods suffer by dynamically increasing the load on the collector. This is a result of the natural robustness of our non-parametric sampling method: when the underlying traffic composition changes, those changes are reflected in each sampled subepoch causing the volume of data reported for each subepoch to increase to ensure steady accuracy.

The sketch-based methods in ElasticSketch and Newton, on the other hand, are limited by the static table sizes configured for the first 20 epochs: once the traffic composition changes, these tables become saturated and excessive hash collisions lead to F1 scores below 0.5. We confirm that the average number of hash collisions per epoch jumps by $2\times$ when the traffic distribution changes in epoch 21. We note that these sketch-based methods also offer no easy way to estimate the accuracy of returned results, so while an operator may become suspicious due to the slight increase in load on the collector, they would have no way to verify that the accuracy of these methods is compromised.

Sketchlearn differs from other methods in that it reconstructs flow keys based on data stored in a multi-level sketch.

Sketchlearn guarantees only that it will be able to extract all flows that make up more than $1/c$ of the total traffic where c is the fixed number of columns in the sketch. We confirm that in this trace, the increased number of sources is caused by a large number of small flows (one to two packets). As such, the threshold to be extracted increases, but none of the added flows are able to meet it and so SketchLearn is unable to extract existing as well as new flows with high enough confidence. SketchLearn does associate accuracy estimates with these results so an operator could be notified of this situation, but would have to reload their switch's pipeline with a larger value of c in order to achieve acceptable accuracy.

Overall accuracy-load tradeoff. As in previous efforts [22], we consider the volume of data returned from switch hardware to the collector (*i.e.*, load on the collector) as a critical resource. Each approximation method can reduce this load while reducing accuracy of query results, leading to a performance curve in accuracy vs. load space. To empirically estimate this curve, we determine several different parameterizations of each method, execute the method with each parameterization over all epochs of the trace, then compute the accuracy and load on collector in each epoch. For some queries the sketch-based methods must export their full sketches to the collector so we report load in terms of both tuples (the number of records or events) and bytes (the total size of data). We use the median of each value over all epochs to estimate the empirical performance curves.

(a) DDoS

(b) TCP New Connections

(c) Port Scan

(d) Super Spreader
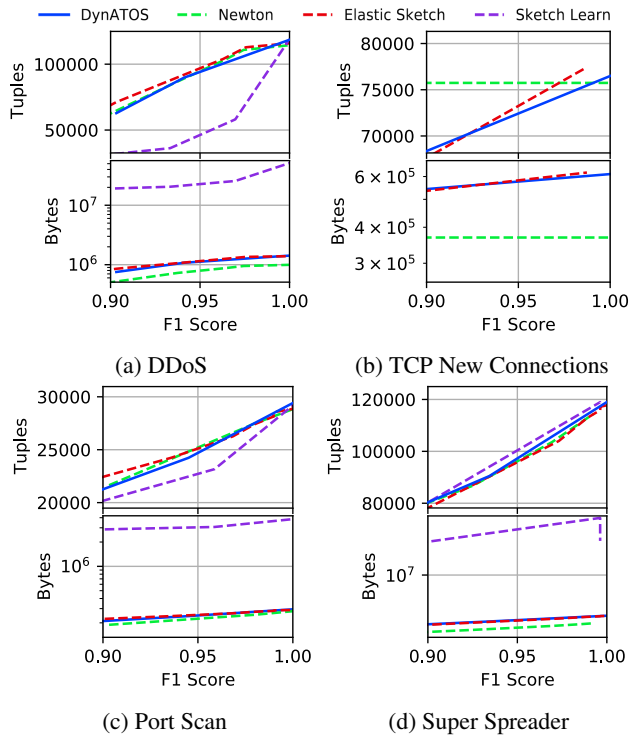
Figure 6: Accuracy vs. overhead curves.

Figure 6 shows performance curves for four different queries with two plots for each query showing overhead as tuples and bytes on the *y*-axis. Here we use the baseline

MAWILab trace so these results represent a mostly static traffic scenario. Note that the lower right-hand corner of these plots is ideal with maximal accuracy and minimal load. We observe that *DynATOS*' novel approximation method (§ 4) performs as well as, if not better than other methods. The sketch-based method proposed by Newton achieves slightly better performance in terms of total data volume on the DDoS and Super Spreader queries because it only sends flow keys from the first distinct operator whereas other methods also return a counter. SketchLearn requires relatively large multi-level sketches to be exported each epoch in order to achieve comparable accuracy on these queries despite it's lower tuple counts. In the case of TCP new connections, we were unable to run a large enough sketch to reach the accuracy range shown here for other methods. We observe that for the TCP new connections query Newton's count-min sketch is highly sensitive to sketch size. For example, adding a single additional counter moves the F1 score across the entire range shown in the plot. *DynATOS*, on the other hand, achieves comparable if not higher performance and offers a wider range of load savings.

6.3 Performance of Scheduling Algorithm

Dynamic query workload. Real telemetry system deployments must deal with dynamics in the number and types of queries submitted to the network over time. Since, to the best of our knowledge, no representative dynamic query workloads are available, we synthesize such workloads based on the following scheme. First, we generate a series of base queries with random aggregation keys and granularities and arrival times based on a Poisson process with rate λ. We suppose these base queries are submitted by a human operator or automated process which then submits followup queries based on base query results. In particular, when each base query terminates, we submit between 0 and 3 followup queries with the same aggregation as the base query, but filters added to select a single aggregation group from the base query's results. For example, if a base query with aggregation key source IP address at 8 bit granularity returned results for 0.0.0.0/8, 10.0.0.0/8, and 192.0.0.0/8, we might submit followup queries to monitor just 10.0.0.0/8 and 192.0.0.0/8. To provide contrasting accuracy and latency goals, base queries are submitted with looser accuracy goals (σ = 100) and latency goals randomly chosen within a range of 1 to 5 epochs, while followup queries are submitted with tighter accuracy goals (σ = 50) and a latency goal of 1 epoch.

Figure 7 shows the evolution of the number of queries submitted by one of our dynamic query workloads (top plot) and traces of different operating metrics (lower three plots). In this workload, the maximum number of queries is submitted in epoch 8 which leads to an infeasible schedule since too many TCAM entries are required to keep track of all filter groups of followup queries. This causes our scheduler to enter fail-safe mode for two epochs to dispatch with the excess queries. Note
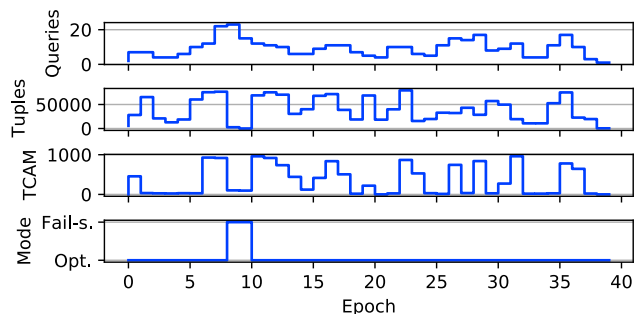
Figure 7: Example time-series of a dynamic query workload (3/5 queries per second).

Figure 8: Performance of *DynATOS* on dynamic query workloads.

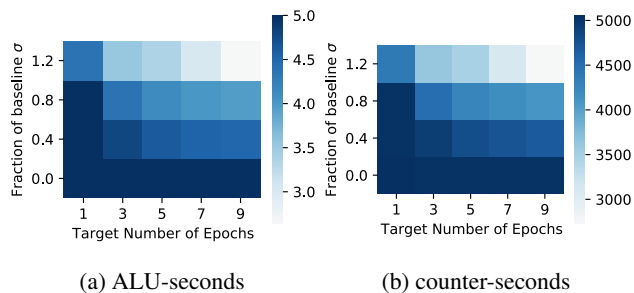(a) ALU-seconds   (b) counter-seconds

Figure 9: Evaluation of median resource usages for selected accuracy (y-axis) and latency (x-axis) targets for a single query. Lighter colors indicate lower resource usages.

that the heuristic algorithm currently used to select queries in fail-safe mode only selects a few queries based on fully disjoint traffic slices leading to reduction of load on collector and TCAM utilization. Under the software-based fail-safe mode mentioned in § 5.2, the load on collector would continue increasing here while TCAM utilization would drop.

To understand how *DynATOS* scales with the rate of dynamic query workloads, we generate a set of five workloads with different base query arrival rates. Figure 8 shows how these different workload intensities affect the performance of *DynATOS* in terms of queries served (Queries), tuples emitted to the collector (Tuples), TCAM entries used (TCAM), epochs spent in fail-safe mode (% Fail-s.), and the percentage of satisfied queries (% Sat.) all per-epoch. We count the number of queries satisfied as the total number of queries that received valid results during the workload run. Note that some queries submitted when the scheduler is in fail-safe mode are denied at submission time allowing an operator to re-submit these queries later. In these experiments we observe that all successfully submitted queries receive results within their target accuracy and latency goals.

We observe that, as expected, the number of queries serviced, load on collector, and number of TCAM entries required all scale linearly with the base query rate. As also expected, the number of queries satisfied decreases as more epochs are spent in fail-safe mode. We observe that the main contributor to infeasible scheduling problems in this scenario is the number of TCAM entries required to satisfy followup queries' filter conditions. We plan to investigate integration of more efficient TCAM allocation algorithms in future work to address this bottleneck.

Relaxation of accuracy & latency goals. Next, we evaluate how our approximation and scheduling method is able to reduce the per-query resource requirements in response to relaxed accuracy and latency goals. We execute the TCP new connections query with varying accuracy and latency goals and measure resource usage over 10 epochs at each setting. Here we report ALU-seconds and counter-seconds which combine both the number of ALUs (or counters) used by the query and the duration for which these resources were used.

Figure 9 show the resulting resource usages as both accuracy and latency goals vary in the form of heatmaps. These
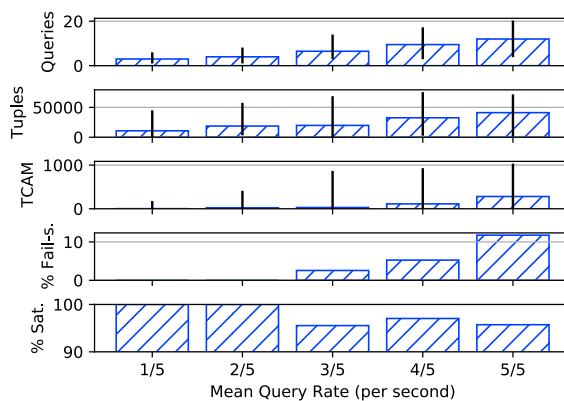
results demonstrate that both accuracy and latency goals can help *DynATOS* leverage our time-division approximation method to reduce resource requirements.

6.4 Scheduling loop overheads

Closed-loop systems like *DynATOS* must quickly gather results and update switch hardware configurations between each subepoch in order to avoid missing potentially critical traffic. We define the inter-epoch latency as the total time spent not waiting for results from switch hardware. In other words, the inter-epoch latency is the total time taken by our system to gather results, reconfigure hardware operations, and decide which operations to execute in the next epoch. We observe two distinct factors that contribute to the inter-epoch latency: the load on the collector and the number of queries installed in switch hardware.

Latency vs. load on collector. The first factor contributing to inter-epoch latency is the volume of data that must be returned and processed after each subepoch. To isolate this effect, we generate synthetic traffic consisting of a certain number of sources each sending a steady stream of packets controlled by a Poisson process. We then run a query that returns a single record for each source so that by varying the number of sources in the traffic, we directly control the number of records returned and hence the load on collector.

Figure 10 shows the distribution of total latency for two different loads. We observe that the median inter-epoch la-

(a) 10 Records   (b) 10^5 Records
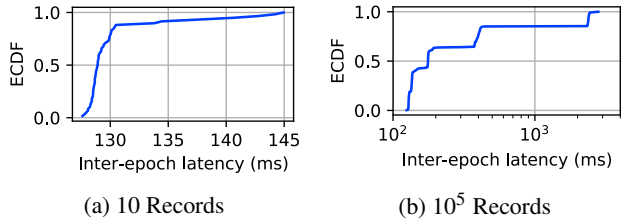
Figure 10: Distribution of inter-epoch latency in our testbed system for different loads on the collector.

tency in both cases is less than 130 ms, but that with higher load the tail latencies grow to over a second. This is likely due to that fact that the collector code must allocate larger memory blocks to process the increased number of tuples returned from the switch. We leave a full investigation of the performance of our software collector to future work.
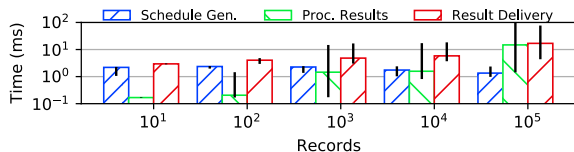
Figure 11: Software overheads as function of tuples exported.

We further investigate how the different components of our query scheduler impact this overall inter-epoch latency by instrumenting the scheduler. Figure 11 shows the latency break down as a function of the number of records processed for three key components: the time to generate a schedule for the epoch (Schedule Gen.), the time spent processing intermediate results at the end of the epoch (Proc. Results), and the time spent sending results back to the query-submitting process (Result Delivery). The results demonstrate that the main variable software latency is the time to process results which scales nearly linearly with the number of records. A more significant bottleneck is imposed by the result delivery time due to the use of a simple REST protocol which requires new TCP connections and data marshaling via JSON. We leave exploration of more efficient IPC mechanisms for this interface to future work.

Latency vs. number of queries. The second main factor contributing to inter-epoch latency is the time required to install and remove query operations on switch hardware. This factor is influenced primarily by the amount of state which must be written into hardware memory which is a function of the number of queries to be installed or removed. We generate synthetic workloads containing different numbers of disjoint queries based again on the TCP new connections query and instrument our switch agent to measure the time taken by writes into hardware memory.

Figure 12 shows the time taken by the hardware writes to add and remove operations (Add Hw. and Remove Hw.) as well as the total time taken by the switch agent (Add Tot. and Remove Tot.) which includes the time to deserialize and validate configurations sent from the collector. These results show that up to 100 queries can be added or removed on our
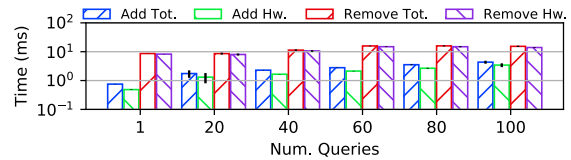
Figure 12: Hardware overheads as function of number of queries.

prototype in $\sim$10 ms (comparable to latencies reported in prior efforts [30, 43]). We also observe that the deserialization and validation conducted by the switch agent imposes minimal overhead. Finally, the total contribution of switch hardware to the overall inter-epoch latency is dominated by operation removal. This is because when removing operations, the switch agent must also reset the entire flow table used by these operations so as to avoid future operations anomalously reporting leftover results.

7   Conclusion and Future Work

Current approaches to telemetry system design struggle to efficiently satisfy dynamism in query workloads and traffic workload composition. By reframing telemetry systems as resource schedulers, in this work, we propose an efficient approximation and scheduling algorithm that exposes accuracy and latency tradeoffs with respect to query execution to reduce hardware resource usage. We evaluate our algorithm by building *DynATOS* and show that our approach is more robust than state-of-the-art methods to traffic dynamics and dynamic query workloads.

While we investigate the common sources of dynamics, both a *horizontal scheduling problem* (*i.e.*, how to design a scheduler to deal with those dynamics for multiple switch hardware stages or multiple distributed switches) and a *vertical scheduling problem* (*i.e.*, incorporation of computing resources, such as stream processing clusters and GPUs—both locally and at remote cloud data centers—into the pool of resources schedulable for telemetry tasks) remain. This opens up a wider question of *where*, not just *when* and *for how long*, telemetry queries should be executed. We plan to investigate this question as part of future work.

Acknowledgments

References

[1] BCM56275 Gb/s Programmable Multilayer Switch Product Brief. https://docs.broadcom.com/doc/56275-PB.

[2] BCM56870 series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56870-series.

[3] COIN-OR Branch-and-cut MIP solver. https://zenodo.org/badge/latestdoi/30382416.

[4] Intel ethernet switch FM6000 series product brief. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/ethernet-switch-fm6000-series-brief.pdf.

[5] Intel Tofino. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch.html.

[6] ONRG: DynATOS. https://onrg.gitlab.io/projects/dynatos/.

[7] Tcpreplay - Pcap editing and replaying utilities. https://tcpreplay.appneta.com/.

[8] Trident3-X4 / BCM56470 Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56470-series.

[9] Kevin Borders, Jonathan Springer, and Matthew Burnside. Chimera: A declarative language for streaming network traffic analysis. In *Proceedings of the USENIX Security Symposium*, pages 365–379, 2012.

[10] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[11] Anne Chao and Chun-Huo Chiu. Species richness: estimation and comparison. *Wiley StatsRef: Statistics Reference Online*, pages 1–26, 2014.

[12] Anne Chao and Chih-Wei Lin. Nonparametric lower bounds for species richness and shared species richness under sampling without replacement. *Biometrics*, 68(3):912–921, 2012.

[13] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, and Ori Rottenstreich. Catching the microburst culprits with snappy. In *Proceedings of the ACM Workshop on Self-Driving Networks*, pages 22–28, 2018.

[14] Xiaoqi Chen, Shir Landau-Feibish, Mark Braverman, and Jennifer Rexford. Beaucoup: Answering many network traffic queries, one memory update at a time. In *Proceedings of the conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 226–239, 2020.

[15] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[16] Cristian Estan, George Varghese, and Mike Fisk. Bitmap algorithms for counting active flows on high speed links. In *Proceedings of the ACM SIGCOMM conference on Internet measurement (IMC)*, pages 153–166, 2003.

[17] Romain Fontugne, Pierre Borgnat, Patrice Abry, and Kensuke Fukuda. MAWILab: Combining Diverse Anomaly Detectors for Automated Anomaly Labeling and Performance Benchmarking. In *Proceedings of the ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2010.

[18] Sriharsha Gangam, Jaideep Chandrashekar, Ítalo Cunha, and Jim Kurose. Estimating TCP latency approximately with passive measurements. In *Proceedings of the International Conference on Passive and Active Measurement (PAM)*, pages 83–93. Springer, 2013.

[19] Sriharsha Gangam, Puneet Sharma, and Sonia Fahmy. Pegasus: Precision hunting for icebergs and anomalies in network flows. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, pages 1420–1428, 2013.

[20] Shahabeddin Geravand and Mahmood Ahmadi. Bloom filter applications in network security: A state-of-the-art survey. *Computer Networks*, 57(18):4047–4064, 2013.

[21] Nicholas J Gotelli and Robert K Colwell. Estimating species richness. *Biological diversity: frontiers in measurement and assessment*, 12:39–54, 2011.

[22] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *Proceedings of the conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 357–371, 2018.

[23] Qun Huang, Xin Jin, Patrick PC Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. Sketchvisor: Robust network measurement for software packet processing. In *Proceedings of the conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 113–126, 2017.

[24] Qun Huang, Patrick PC Lee, and Yungang Bao. Sketch-learn: Relieving user burdens in approximate measurement with automated statistical inference. In *Proceedings of the conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 576–590, 2018.

[25] Qun Huang, Haifeng Sun, Patrick PC Lee, Wei Bai, Feng Zhu, and Yungang Bao. Omnimon: Re-architecting network telemetry with resource efficiency and full accuracy. In *Proceedings of the conference of the ACM Special Interest Group on Data Communication (SIG-COMM)*, pages 404–421, 2020.

[26] Teng Li, Jian Tang, and Jielong Xu. Performance modeling and predictive scheduling for distributed stream data processing. *IEEE Transactions on Big Data*, 2(4):353–364, 2016.

[27] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. Nitrosketch: Robust and general sketch-based monitoring in software switches. In *Proceedings of the conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 334–350. 2019.

[28] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 101–114. ACM, 2016.

[29] Sharon L Lohr. *Sampling: Design and Analysis: Design And Analysis*. CRC Press, 2019.

[30] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. DREAM: Dynamic resource allocation for software-defined measurement. *ACM SIG-COMM Computer Communication Review*, 44(4):419–430, 2014.

[31] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. SCREAM: Sketch resource allocation for software-defined measurement. In *Proceedings of the ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, page 14, 2015.

[32] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Trumpet: Timely and precise triggers in data centers. In *Proceedings of the conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 129–143, 2016.

[33] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. In *Proceedings of the conference of the ACM Special Interest Group on Data Communication (SIG-COMM)*, pages 85–98, 2017.

[34] Boyang Peng, Mohammad Hosseini, Zhihao Hong, Reza Farivar, and Roy Campbell. R-storm: Resource-aware scheduling in storm. In *Proceedings of the Annual Middleware Conference*, pages 149–161, 2015.

[35] Hooman Peiro Sajjad, Ken Danniswara, Ahmad Al-Shishtawy, and Vladimir Vlassov. Spanedge: Towards unifying stream processing over central and near-the-edge data centers. In *Proceedings of the IEEE/ACM Symposium on Edge Computing (SEC)*, pages 168–178, 2016.

[36] Anshu Shukla and Yogesh Simmhan. Model-driven scheduling for distributed stream processing systems. *Journal of Parallel and Distributed Computing*, 117:98–114, 2018.

[37] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Distributed network monitoring and debugging with SwitchPointer. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 453–456, 2018.

[38] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 561–575. ACM, 2018.

[39] Minlan Yu, Lavanya Jose, and Rui Miao. Software defined traffic measurement with OpenSketch. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 29–42, 2013.

[40] Lihua Yuan, Chen-Nee Chuah, and Prasant Mohapatra. ProgME: towards programmable network measurement. *IEEE/ACM Transactions on Networking*, 19(1):115–128, 2011.

[41] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J Freedman. Live video analytics at scale with approximation and delay-tolerance. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 377–392, 2017.

[42] Haiquan Chuck Zhao, Ashwin Lall, Mitsunori Ogihara, Oliver Spatscheck, Jia Wang, and Jun Xu. A data streaming algorithm for estimating entropies of OD flows. In *Proceedings of the ACM SIGCOMM conference on Internet measurement (IMC)*, pages 279–290, 2007.

[43] Yu Zhou, Dai Zhang, Kai Gao, Chen Sun, Jiamin Cao, Yangyang Wang, Mingwei Xu, and Jianping Wu. Newton: Intent-driven network traffic monitoring. In *Proceedings of the ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, pages 295–308, 2020.

A Appendix: Application of Cluster Sampling

In this section, we discuss details of key equations enabling our scheduling approach's accuracy and latency tradeoffs. To maintain a self-contained discussion, some content is repeated from § 4.

A.1 Trading Off Accuracy

Given fixed scheduling epochs, we can trade off accuracy for reduced resource requirements by sampling a subset of the subepochs in which to execute a particular query. We leverage cluster sampling theory [29] to expose this tradeoff while maintaining accuracy goals. To simplify our discussion, we first consider the case where a query is executed in a single epoch and then expand to the case where a query is executed across multiple epochs.

Single Epoch Case. Consider the case where a particular query executes in n of the N total subepochs. Our goal is to estimate the value that would have resulted from running the query in all subepochs based only on these n subepoch results. First, we note that each subepoch defines a cluster of packets that traverse the switch during that subepoch. Next, since each query executes over every packet of the subepochs in which it is scheduled, we note that the subepoch results represent a sample of n of the N total subepoch clusters. To ensure that each subepoch has an equal probability of being sampled by a particular query, we shuffle subepochs prior to execution. Cluster sampling theory [29] then lets us estimate the results of these queries over the entire N subepochs as well as the error of this estimator based on the variance between the observed subepochs. For example, we can estimate a query that maintains a sum by

$$\hat{t} = \frac{N}{n} \sum_{i \in S} t_i$$

which has standard error

$$SE(\hat{t}) = N \sqrt{\left(1 - \frac{n}{N}\right) \frac{s_t^2}{n}}$$

where S is the index set of which subepochs have been sampled, t_i is the query's result in the i-th subepoch, and s_t^2 is the sample variance of the t_i's. Clearly, executing a query for fewer subepochs leads to greater sampling error while executing a query in each subepoch leads to zero sampling

error. This equation also shows that, if n is set to a fixed ratio of N, error grows as a function of N so we do not expect to increase accuracy by dividing epochs into larger numbers of subepochs. Corresponding theory and equations exist for other update operations such as averages and extreme values.

Multiple Epoch Case. Due to changing traffic distributions or large query workloads, a query may not be able to fulfil its accuracy goal in a single epoch and the scheduler must form results based on the estimates from multiple epochs. Considering again the sum example, let $t_{i,j}$ be the query's result in the i-th subepoch of the j-th epoch, n_j be the number of subepochs in which the query executed in the j-th epoch, and E be the total number of epochs in which the query is executed. By the self-weighting property of $\hat{t}$, we can take a simple mean of the $\hat{t}_j$'s to get an unbiased estimator of the query's result over the E epochs,

$$\hat{t}_E = \frac{1}{E} \sum_{j=1}^{E} \hat{t}_j = \frac{1}{E} \sum_{j=1}^{E} \frac{N}{n_j} \sum_{i \in S_j} t_{i,j} \tag{5}$$

which has standard error

$$SE(\hat{t}_E) = \frac{N}{E} \sqrt{\sum_{j=1}^{E} \left(1 - \frac{n_j}{N}\right) \frac{s_{t_j}^2}{n_j}} \tag{6}$$

because subepochs are chosen independently in each epoch (*i.e.*, the sampled index sets S_j, which are the only random variables in this formulation, are independent).

Application to Scheduling. Our system uses the point estimates provided by Equation 5 to calculate estimated query results. We also utilize Equation 6 for two purposes: (i) determining when accuracy goals have been fulfilled and (ii) estimating the number of subepochs in which the scheduler must execute particular queries. Since the first item can be evaluated with a simple threshold check, the rest of this section explains the second item. We assume that each query executes a single update operation (*e.g.*, a sum) in its reduction and note that multiple operations could be expressed in multiple queries.

Note that for a given standard error target ($SE(\hat{t}_E) = \sigma$) we can rearrange Equation 6 to solve for the number of subepochs that must be sampled in the E-th epoch as follows,

$$n^{acc} = \frac{s_{t_E}^2 N^2}{E^2 \sigma^2 - \left(\sum_{j=1}^{E-1} \text{Var}(\hat{t}_j)\right) + N s_{t_E}^2} \tag{7}$$

Given a query's target standard error σ, observed values of $s_{t_j}^2$ and n_j from prior epochs, and an estimate of $s_{t_E}^2$ (based on the $s_{t_j}^2$'s), we can use Equation 7 to determine a lower bound on the number of subepoch in which a query should execute. Note that if $\sigma = 0$, then $n^{acc} = N$ and the query will be executed in all of the subepochs in its first epoch. As σ

increases, n^{acc} decreases freeing more of the subepochs for other queries. For example, Figure A.1a shows the result of evaluating Eq. 7 for the first epoch of a query, indicating that if the accepted standard error is large enough, the scheduler only needs to execute the query in a single subepoch.

Limitations. We note that Equation 7 can become unstable when the accuracy goal σ cannot be obtained in a single epoch given the results of prior epochs. This condition results when $E^2\sigma^2 + Ns_{t_E}^2 \leq \sum_{j=1}^{E-1} \text{Var}(\hat{t}_j)$ causing the value of n^{acc} to be negative or undefined. Moreover, when n^{acc} is negative, its magnitude has the wrong sense with respect to σ: smaller (tighter) values of σ reduce the magnitude of n^{acc}. Rather than dropping the query, we make a maximum allocation choice ($n^{acc} = N$) and retain the query for future epochs until its accuracy target is met. So long as $\text{Var}(\hat{t}_j) < \sigma^2$ for enough of those future epochs, n^{acc} will eventually stabilize.
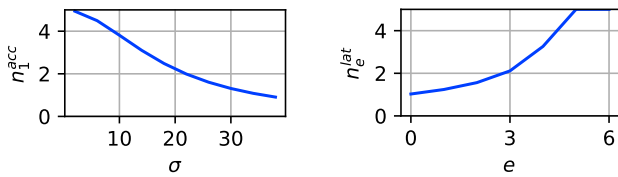
(a) Increasing σ reduces n^{acc} in the first epoch.

(b) n^{lat} increases as deadline $E = 6$ approaches.

Figure A.1: Numeric evaluations of Eqs. 7 and 8 assuming fixed variance $s_t^2 = 8$, $N = 5$, and queries get $3/5^{th}$ of the subepochs.

A.2 Trading Off Latency

In addition to the accuracy tradeoff discussed above, we can tradeoff result latency for reduced resource requirements by executing a query's operations across several epochs. The key observation enabling this tradeoff is that by spreading the sampled subepochs over several epochs, the query can reduce its per-epoch requirements while still attaining its accuracy goal. Operators leverage this tradeoff by specifying larger latency goals on queries which do not require fast returns. We then adapt Equation 6 to estimate how many subepochs should be executed in the current epoch based on both past and anticipated future results.

Accounting for Past and Future Results. Under the latency tradeoff, we approach the problem of determining how many subepochs to execute from the perspective of the point in the future when the query completes. At this point Equation 5 will be used to estimate the query's result and Equation 6 must satisfy the query's accuracy goal. Moreover, assuming we are satisfying the query's latency goal, E is equal to the target number of epochs.

Now we consider the task of estimating the number of subepochs to execute during some epoch e before the query's final epoch E. Note that the sum in Equation 6 can be split

around epoch e into a past component

$$past = \sum_{j=1}^{e-1} \left(1 - \frac{n_j}{N}\right) \frac{s_{t_j}^2}{n_j}$$

and a future component

$$future = \sum_{j=e+1}^{E} \left(1 - \frac{n_j}{N}\right) \frac{s_{t_j}^2}{n_j}.$$

We can then directly adapt Equation 7 to provide the required number of subepoch in epoch e accounting for both past and future components as

$$n^{lat} = \frac{s_{t_E}^2 N^2}{E^2\sigma^2 - N^2(past + future) + Ns_{t_E}^2} \quad (8)$$

Figure A.1b shows the result of evaluating Equation 8 in each epoch leading up to a query's target latency of $e = 6$ assuming that the operation gets $3/5^{th}$ of the number of subepochs requested in each epoch. Since in this case, the query is not given its full requested number of subepochs, the target n^{lat} increases dynamically to meet the deadline. This indicates that Equation 8 can dynamically drive scheduling decisions even when its results are not taken literally in each epoch (as may be the case when multiple queries compete for resources).

Limitations. Equation 8 faces the same issues as Equation 7 in that it may still be infeasible to satisfy σ given past results and the anticipated gains of future results. In such cases we again take $n_j = N$ and count on gaining sufficient resources in future epochs to satisfy the accuracy goal. To understand the dynamics of this decision, Figure A.2 shows the relation between target and actual number of epochs for a number of accuracy goals. We assume here that queries anticipate getting $3/5^{th}$ of the subepochs, actually receive $3/5^{th}$ of what they ask for, and all other settings are as in Figure A.1. As can be seen when the accuracy target is too tight (*e.g.*, $\sigma = 6$) executing in less than a certain number of epochs ($e = 5$) is infeasible and the query's latency goal cannot be met.
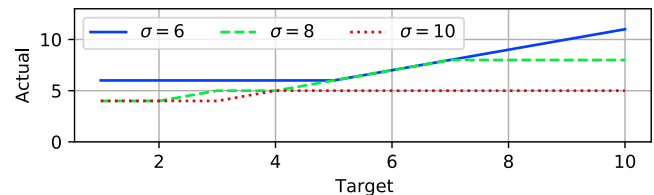
Figure A.2: Relation between target number of epochs and the actual required number of epochs.

A.3 Correcting Distinct Operators

Many useful queries also involve identifying and counting distinct elements. We consider the particularly prominent

query structure where the results of a distinct operator are fed through a reduce operator with a slightly coarser granularity key. For example the DDoS query considered in § 2.1 contains two main stateful operators: (i) finding distinct source, destination pairs and (ii) reducing with destination as the key to count the number of unique sources contacting each destination. The key problem is that, while the methods above provide sound estimators for packet and byte counts, they do not correct for elements which may have been entirely missed in the distinct operator due to sampling. Such errors lead to a downward bias on distinct counts based on sampling which could cause key events like DDoS attacks to go unnoticed. To correct for this source of error, we leverage the Chao estimator without replacement when performing reductions after distinct operators impacted by sampling. Chao estimators [11, 12] are commonly used by "species richness" studies in the biological sciences to solve a related type of distinct count problem [21]

This estimator is given by

$$\hat{S}_{Chao1,wor} = S_{obs} + \frac{f_1^2}{\frac{n}{n-1}2f_2 + \frac{q}{1-q}f_1} \tag{9}$$

where S_{obs} is the number of elements observed in the sample, f_1 is the number of elements observed only once, f_2 is the number of elements observed only twice, n is the total number of elements in the sample, and q is the sampling rate. To use this estimator, we modify distinct operators executed in the data plane to additionally count the number of packets observed for each distinct element (essentially transforming them into normal count reduction operators). After gathering results, we can then easily compute the inputs required by Equation 9. Note that the variance of $\hat{S}_{Cha1,wor}$ can also be easily obtained from the same information as shown in the original description of this estimator [12], providing network operators with approximate accuracy of these results as well.

HeteroSketch: Coordinating Network-wide Monitoring in Heterogeneous and Dynamic Networks

Anup Agarwal
Carnegie Mellon University

Zaoxing Liu
Boston University

Srinivasan Seshan
Carnegie Mellon University

Abstract

Network monitoring and measurement have always been critical components of network management. Recent developments in sketch-based monitoring techniques and the deployment opportunities arising from the increasing programmability of network elements (e.g., programmable switches, Smart-NICs, and software switches) have made the possibility of accurate, detailed, network-wide telemetry tantalizingly within reach. However, the wide heterogeneity of the programmable hardware and dynamic changes in both resources available and resources needed for monitoring over time make existing approaches to network-wide monitoring impractical.

We present HeteroSketch, a framework that consists of two main components: (1) a profiling tool that automatically quantifies the capabilities of arbitrary hardware by predicting their performance for sketching algorithms, and (2) an optimization framework that decides placement of measurement tasks and resource allocation for devices to meet monitoring goals while considering heterogeneous device capabilities. HeteroSketch enables optimized deployments for large networks (> 40,000 nodes) using a novel clustering approach and enables prompt responses to network topology, traffic, query, and resource dynamics. Our evaluation shows that HeteroSketch reduces resource overheads by $20-60\%$ compared to prior art, while maintaining monitoring performance, coverage, and accuracy.

1 Introduction

The ability to monitor network traffic in-situ and at-large-scale is a critical enabler for many networked management applications, including traffic engineering, load balancing, attack and anomaly detection, provisioning, and congestion control/fairness [1–7]. However, network-wide monitoring has proven to be challenging due to limitations on what measurements can be made and where these measurements can be taken. Recent developments in sketch-based monitoring and network programmability have made significant progress in eliminating these limitations and have made it possible to consider practical network-wide monitoring designs.

Sketch-based monitoring designs [8–13] demonstrate that sketches offer provable accuracy guarantees on a wide spectrum of metrics of interest using a small amount of memory and that independent sketch instances monitoring different parts of the network can be merged to obtain network-wide aggregated results. As a result, sketch-based monitoring has emerged as a promising alternative to traditional sampling-based monitoring tools (e.g., NetFlow [14] and sFlow [15]). The growing popularity of programmable network elements, such as programmable switches [16, 17], SmartNICs [18, 19], and software-switches [20, 21], have made it possible to deploy these sketch-based designs throughout a network – enabling highly-effective network-wide monitoring capabilities.

Despite significant recent progress [10–13, 22], we argue that deploying sketch-based monitoring in a network-wide setting remains impractical. The reason behind this is that existing network-wide solutions [11, 22, 23] assume an abstract network model without properly considering the *heterogeneity* and *dynamics* in the network. First, with growing types of programmable devices whose hardware architectures are dramatically different (e.g., ASIC, CPU, FPGA), it remains unclear how to deploy sketches among heterogeneous computation and memory hierarchies for optimized resource efficiency. Second, since monitoring capabilities share the same infrastructure with other network services [24–26] and monitoring requirements vary over time, the available and required resources for monitoring can change dynamically. We require an agile solution that can incorporate device heterogeneity and quickly adjust to network dynamics for best possible monitoring performance.

In this paper, we present **HeteroSketch**, a network-wide flow monitoring framework that coordinates sketch-based measurement to determine task placement and resource allocation for a network of heterogeneous devices. HeteroSketch has two main components: (1) a device characterization tool that automates quantified reasoning about the performance and resource usage of sketches on arbitrary de-

vice architectures and (2) an optimization framework that computes the placement of measurement tasks while considering available heterogeneous device resources and monitoring goals including forwarding performance, resource efficiency, monitoring accuracy, and flow coverage.

When designing our device characterization tool, we need to deal with a broad spectrum of programmable architectures such as CPU [20, 21], FPGA [19, 27], ASIC [16], and multi-core system-on-chip (SoC) [18]. Given the difficulty in accurately predicting the performance of arbitrary code under diverse workloads and hardware architectures [28, 29], we scope our efforts to sketches to create a practical solution. Our design is inspired by the observation that many sketches perform similar computations. We analyze the key operations of sketches and find that the performance of a sketch depends heavily on primitive operations – hash computations, counter updates, and random memory lookups. With that in mind, we then characterize these operations on current (and possibly new) hardware using automated micro-benchmarks, and leverage these measurements to express the performance and resource usage of a sketch. As evaluated in §7.1, our profiler accurately predicts sketch performance on programmable hardware with less than 6% mean relative error.

With precise performance profiles as input, HeteroSketch must address the tightly coupled monitoring goals, traffic demands, forwarding performance, sketch configurations, and resource usage, and the tradeoffs between them. Task placement and resource allocation requires a carefully crafted optimizer to incorporate the cost/benefit of different deployment options. We formulate a Mixed-Integer Program (MIP) to optimize resource efficiency while preserving forwarding performance, monitoring accuracy, and flow coverage.

Given the complexity introduced due to heterogeneous devices (non-convexity) and network scale, even a state-of-the-art solver [30] takes hours to solve the MIP, leading to stale solutions in face of network dynamics. We develop a clustering technique based on observed structures in network topologies and traffic patterns to partition the optimization into independent sub-problems. This allows HeteroSketch to scale to today's data center networks having tens of thousands of devices and respond to dynamics within a few seconds to a few minutes while maintaining near optimal allocations.

We implement HeteroSketch by porting state-of-the-art sketch implementations [8, 9, 11, 13] into representative programmable devices (Barefoot Tofino [16], Netronome Agilio SmartNIC [18], Xilinx FPGA NIC [27], and Open vSwitch (OVS) [20]) and encode the optimization in the Gurobi [30] solver. Our evaluation with more than 40,000 nodes demonstrates that our heterogeneity-aware optimization can achieve 20 − 60% better resource efficiency (e.g., 50k CPU cores instead of 70k CPU cores) compared to prior solutions with the same performance, accuracy, and coverage while responding to network dynamics in a few seconds to a few minutes.

Contributions. We make the following contributions:

- We present HeteroSketch, the first system to our knowledge that performs coordinated sketch-based network-wide monitoring over a network of heterogeneous devices and caters to network dynamics. (§3)

- We develop a profiler that allows users to predict the performance of sketches on heterogeneous devices. (§4)

- We formulate a mixed-integer program to optimize sketch placement and resource allocation over heterogeneous devices and propose techniques to quickly optimize when topology, queries, traffic, or resources change. (§5, §6)

- We show that HeteroSketch is able to place tasks and allocate resources over network topologies, achieving greater scale and optimality than existing systems. (§7)

2 Background and Motivation

In this section, we describe how heterogeneous programmable data planes bring new opportunities and challenges to deploy network-wide monitoring under varying demands. We then discuss existing network-wide monitoring efforts.

Programmable Data Plane. Progress in programmable network devices is moving the network data plane towards a highly programmable infrastructure. This programmable infrastructure opens up the opportunity to develop measurement algorithms for a variety of fine-grained, flexible measurement tasks. For example, significant progress has been made in developing sketching algorithms [11, 12, 31] on the RMT architecture [32], where packets are processed over a series of reconfigurable match-action tables with user-defined actions in a pipeline. Similarly, multi-engine SmartNICs [18] consisting of a pool of general purpose processing elements (i.e., micro-engines) are a cost-effective option to allow hosts to offload monitoring capabilities or other parallel computation from CPU. These programmable devices enable the development of highly flexible and performant future-proof network-wide monitoring for various network demands.

New Requirements in Network-wide Monitoring. For a long time, network monitoring research has been focused on pursuing *accuracy* over other goals, such as forwarding *performance* with less computing and memory resources, and *scalability* by supporting larger-scale networks. While these requirements continue to be important, we believe that there are two significant roadblocks that make it difficult or impractical to use existing sketch-based designs at scale in future programmable data planes:

- *Heterogeneity in the network:* Network data planes are becoming increasingly heterogeneous with devices such as x86-based software switches [20], ASIC-based programmable switches [16], multicore system-on-chip (SoC) SmartNICs [18], and FPGA NICs [19, 27]. These devices are designed with diverse architectures and present very different resource bottlenecks to the programs that execute on them. The challenge lies in how to precisely character-
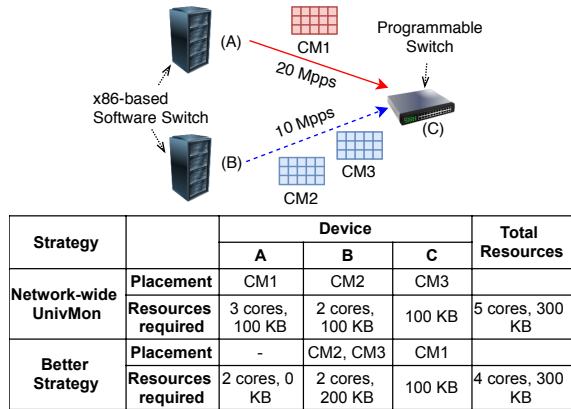
Figure 1: Example of network-wide UnivMon not optimally placing the sketches.

Strategy		Device			Total Resources
		A	**B**	**C**	
Network-wide UnivMon	Placement	CM1	CM2	CM3	
	Resources required	3 cores, 100 KB	2 cores, 100 KB	100 KB	5 cores, 300 KB
Better Strategy	Placement	-	CM2, CM3	CM1	
	Resources required	2 cores, 0 KB	2 cores, 200 KB	100 KB	4 cores, 300 KB

ize the performance of monitoring programs on current and possibly new devices and use these insights to optimize resource usage.

- *Network dynamics:* Network monitoring serves as a data collector and analyzer for other co-located network services (e.g., traffic engineering, load balancing, and anomaly detection). As all these services share the same infrastructure and their monitoring needs change, any network-wide monitoring system should quickly adjust to the following "network dynamics": (1) topology change, (2) monitoring query change, (3) traffic demand change, and (4) available resource change, in order to provide best monitoring performance and not interrupt other concurrent services.

Current Network-wide Monitoring and Limitations. In small networks, we can consider using techniques that record all packets or flows passing through the network for full accuracy (e.g., T-RAT [33], vCRIB [34], and OmniMon [35]). However, in practice, with the desire for *real-time* and *accurate* monitoring over large traffic volumes and dynamics, operators usually cannot afford to record all packets or flows on their network devices due to high resource usage and processing latency. Recently developed zoom-in techniques (e.g., Sonata [36] and ProgME [37], and others [38, 39]) provide no theoretical accuracy guarantee for dynamic traffic workloads. Systems such as NetFlow/sFlow [14, 15] and cSAMP [23] reduce overhead by recording only a fraction of packets via packet or flow sampling to compute measurement results. As shown in prior efforts [40, 41], these sampling approaches have low measurement accuracy in various tasks and workloads.

Sketching algorithms (sketches) address the drawbacks of sampling. At a high level, sketches [8, 9, 42] are probabilistic data structures that store a small summary of the input traffic. They allow a proven trade-off between the *accuracy* of supported queries and the *space* of the summary. Sketching techniques have efficiently supported various monitoring tasks including: heavy hitter detection [8, 9, 11, 12, 31, 43], traffic change detection [11, 44], anomaly detection [11, 22, 45],

Scheme	Resource Overhead	Heterogeneity & Dynamics	Memory-Accuracy Tradeoff
cSAMP [23]	High	✗	Poor
vCRIB [34]	High	Limited	Poor
OmniMon [35]	High	Limited	Poor
UnivMon [11]	Medium	✗	Good
HeteroSketch	Low	✓	Good

Table 1: Summary of qualitative comparisons of existing schemes and our approach (HeteroSketch).

entropy estimation [11, 46, 47], counting distinct flows [11, 12]. Most sketches mentioned above can be *linearly merged* to obtain aggregated results with the same additive error guarantees [48]. For example, Sketch 1 measuring flow set A can be merged with Sketch 2 measuring flow set B (e.g., by addition of the two counter tables) to obtain statistics about a combined flow set $A \cup B$, as long as Sketches 1 and 2 share the same hash and memory configurations.

Unfortunately, existing sketch-based monitoring solutions don't consider heterogeneity and dynamics, which affects their resource efficiency and/or accuracy (Table 1). For instance, Figure 1 shows a simple scenario where network-wide UnivMon [11] does no optimally place Count-Min Sketches [8], resulting in using 5 cores instead of 4. In this setting, we have three devices (CPU A, CPU B, and programmable switch C). We want 1 Count-Min sketch (CM1) to monitor traffic between devices A and C, and we want 2 Count-Min sketches (CM2 and CM3) to monitor traffic between devices B and C. We also assume that the programmable switch can only fit one sketch in its share of switch resources for monitoring tasks. CPU A requires 2 cores for forwarding 20Mpps while CPU B requires 1 core for 10Mpps. The key decision is which sketch should go on the programmable switch. *Better Strategy:* If we place CM1 on the switch, both CM2, CM3 run on CPU B consuming 1 core for sketching (combined 20M sketch operations per second). *Network-wide UnivMon:* If we place CM2 (or CM3) on the switch, then CM1 must be placed on CPU A consuming 1 core for sketching and CM3 (or CM2) must be placed on CPU B again consuming 1 core for sketching, for a total of 2 sketching cores. Note, placing only one of CM2 or CM3 on CPU B consumes 1 sketching core as cycles are wasted busy-polling for packets for performance reasons. UnivMon produces this placement as it tries to balance memory load across devices.

3 System Overview

We describe the high-level design of HeteroSketch and highlight the key challenges that the design must address.

3.1 Problem Scope

HeteroSketch provides a "One Big Switch" abstraction to the user, wherein the user can specify monitoring require-
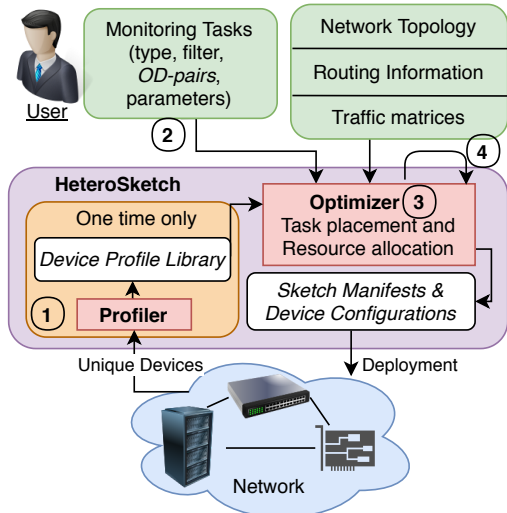
Figure 2: HeteroSketch Overview.

ments over all or a subset of traffic flowing between network endpoints or *origin-destination (OD)* pairs. Monitoring requirements include the type of the sketch, flow filter, and accuracy requirement (Figure 2). HeteroSketch takes these monitoring requirements and assigns particular sketch-based tasks among the components of a heterogeneous network of diverse devices using state-of-the-art sketching algorithms such as Count-Min [8], Count Sketch [9], and UnivMon [11].

This abstraction can be used to manage network monitoring in various settings. For example, HeteroSketch could be applied by Internet Service Providers (ISPs) to manage monitoring services internally or for their clients. In this paper, we envision cloud providers being early adopters of our network-wide monitoring system. In a multi-tenant cloud environment, a cloud provider would be able to offer monitoring-as-a-service to the tenants. Tasks corresponding to queries from different tenants would be placed within the network as opposed to just on the end-hosts that the tenant is using, i.e., NIC and switch resources can be indirectly accessed by tenants for monitoring purposes. The operator would make control decisions using a centralized view to address different measurement objectives. For instance, the operator could (1) manage monitoring requirements submitted by multiple independent tenants while incorporating any potential contention between monitoring tasks placed on the same device; and/or (2) deploy their own monitoring tasks (e.g., detecting compromised tenant VMs [49]). In any of the scenarios, our system makes it possible to load balance monitoring tasks between servers, NICs, and switches, and to prioritize resources such as CPU cores for other critical services and cloud applications.

3.2 HeteroSketch Workflow

As depicted in Figure 2, HeteroSketch has two main components: a performance profiler and an optimization framework. **Performance Profiler (①) in Figure 2):** For any new device,

HeteroSketch needs to conduct offline performance characterization to add a new abstract profile into its device profile library. The device profile library allows HeteroSketch to predict the performance-resource trade-offs of different sketch configurations. We describe this Profiler in detail in Section 4.
Optimization Framework: Once HeteroSketch obtains user input as ② (monitoring requirements), the HeteroSketch Optimizer ③ outputs the configuration and mapping of sketches to devices (i.e., *sketch manifests* for each device) and the resources allocated to each device (i.e., *device configuration*). Based on the Optimizer output, HeteroSketch deploys sketches into the network and gathers network-wide statistics as other monitoring systems. If there are dynamic changes in user input, network topology, traffic demands, and available resources, HeteroSketch will perform a quick re-optimization ④. We describe this Optimizer in detail in Section 5.
Supported Queries: HeteroSketch currently supports sketch-based flow-level telemetry queries over flow sets defined over OD-pairs; e.g., heavy hitters, flow changes, entropy, distinct flows, among others, over flows across one or more OD pairs. For instance for network-wide heavy-hitters, a user can specify multiple OD-pairs to be monitored for the same query. HeteroSketch will then instantiate and collect (linearly merge) data from multiple instances of sketches (e.g., Count-Min and UnivMon) while ensuring that all OD-pairs are monitored and resource overhead is minimized. That said, telemetry queries that are not defined over flows, such as path-level queries in In-band Network Telemetry (INT) [50], or packet-level queries, which are not supported by sketches, are outside the scope of this paper.

3.3 Challenges and Key Insights

We describe the three main challenges that our design faces.
C1: [Heterogeneity] Predicting sketch performance for different resource allocations. Optimizing resource utilization requires characterization of exact costs and benefits of different deployment configurations. This is challenging because many characteristics of the program and the device architecture impact the processing time per packet.[1] For instance, devices may execute certain operations using ASICs and others using general purpose cores. The time for an instruction might depend on the allocated resources, e.g., memory access time depends on the working set. The program might have a complex, unpredictable control flow with many data dependencies. Past systems [28, 29] rely on low-level architecture specific counters and cache analysis to provide performance estimates. Such approaches do not provide the accuracy needed and would be difficult to generalize to other hardware.
    Insight: We observe that (1) the primitive operations of sketches (e.g., hashing, memory updates) largely determine

---

[1]We represent performance in terms of time per packet or inverse throughput and use these terms interchangeably.

packet processing time, and (2) based on data flow analysis, most sketches have limited data dependencies and limited control flow. This means that there are enough independent operations to be performed (either for the same packet or across packets), that performance is broadly determined by the number of operations rather than their inter-dependencies. Therefore, we design a benchmark suite consisting micro-benchmarks of primitive sketch operations for a small set of sketch manifests. The Profiler composes these benchmarks to generate algebraic characterization of the resource-performance trade-off for arbitrary sketch manifests.

C2: [Formulation & Scalability] The optimization formulation of sketch placement over heterogeneous hardware results in a large and complex NP-Hard optimization problem. Despite using a state-of-the-art commercialized solver (e.g., Gurobi), it still needs order of hours to finish even for relatively small networks (≈1000 nodes). Specifically, the complex *non-convex* device profiles and scale of the network slow down the optimization.

Insight: We use the solver's advanced features (bi-linear constraints) to incorporate the non-convex device profiles. For scalability, we partition the optimization problem into disjoint sub-problems which are solved concurrently. We define these sub-problems by partitioning the network into clusters. We find that traditional clustering techniques such as spectral clustering [51] either result in infeasible sub-problems or solutions which are far from optimal (see §6). Our key insight is to define clustering affinity between nodes based on OD-pairs (traffic and monitoring requirements) rather than just network structure.

C3: [Dynamics] Sketch manifests and device configurations can become stale due to network dynamics including changes in monitoring requirements, available resources, logical topology, and traffic demands. A robust solution should adapt its deployment at the rate of these changes.

Insight: While the insights in C2 help scale the Optimizer to handle large topologies and bring down solving time from order of hours to a few minutes (§7.2), we supplement the clustering approach with a *Fast Path* that allows quicker responses (in a few seconds) to network dynamics. It leverages our observation that it is sufficient to recompute the placement/configuration for only a subset of devices which are *directly affected* by the network dynamics (§6.2).

4 Performance Profiler

We leverage the common structure of sketches to make the performance prediction problem tractable. As an example, we describe the structure of a canonical Count-Min sketch [8] that can be used for maintaining a summary of per-flow sizes.

The sketch maintains a counter table of rows and columns. On observing a <*key*, *value*> pair, a hash function is computed over the *key* for each row of the sketch. These hash values are used to index into the rows and the content of the corresponding cells is incremented by *value*. The updates to different rows are completely independent of each other, which is similar for a large set of sketches [8, 9, 11, 13, 42, 52]. With this structure, hash computation and memory update operations consume the majority of the time.

Our Approach. While the common structure of sketches allows us to manage the complexity introduced from the program's side, we still need to manage the complexity due to diverse devices. Specifically, for each device type, we have a three-phase approach to determine the sketch performance:

- **Phase 1:** Measure the time for primitive operations.
- **Phase 2:** Compose the time for different operations.
- **Phase 3:** Consider impact of device configurations.

Before diving into the details of the three phases, we provide a brief overview of the Profiler's operation and its setup.

Setup: The Profiler uses a three-device testbed consisting of the device being studied (or device under test, DUT), a sender, and a receiver. The three devices connected in a linear topology with the DUT configured to forward traffic from the sender to the receiver. Such a setup can be created without a lab environment or re-wiring, by changing forwarding configuration in a local or cloud deployment. A more detailed description of this testbed is provided in §7.1.

Overview: The Profiler treats the devices as "black-boxes" and makes few assumptions about the architecture. We assume that the DUT has a library to implement sketch manifests and that it exposes an API to allocate resources. The Profiler uses this API to study the DUT's forwarding rate for a limited set of sketch manifest and device configuration combinations. The Profiler does not need any code instrumentation, hardware counters, or precise time-stamping, it simply studies the end-to-end forwarding rate.

For each device, the Profiler models time per packet as an algebraic function of *sketch parameters*, *device parameters* and *device configuration*. The sketch parameters include counts for primitive sketch operations, which are obtained from the sketch manifest (e.g., sketch type, the numbers of rows and columns [8, 9], and the number of levels (sketch instances) [11]). We obtain device parameters from micro-benchmarks for the primitive operations. Device configuration specifies the resources, including memory, processor cores, micro-engines, switch stages/ALUs, lookup tables, flip-flops, and/or DSPs. We believe this approach generalizes to support architectures beyond the hardware at hand (Table 2).

4.1 Detailed Design of the Profiler

Phase 1: Primitive Operations. In this phase, we evaluate the time for the following primitive operations per sketch update: hash computations (compute capabilities), memory accesses (impact of memory hierarchy), coin tosses (random

Type	Hardware
CPU (Open vSwitch)	Intel® Xeon® Silver 4110 CPU @ 2.10GHz (32KB L1, 256KB L2, 8MB L3 cache) with Mellanox ConnectX 4 NIC [53][20]
SoC SmartNIC	Netronome Agilio® CX 1x40GbE
FPGA NIC	Xilinx® Alveo™ U280 Data Center accelerator card
Prog. Switch	Barefoot Tofino

Table 2: Devices tested with Profiler.

(a) CPU   (b) SoC NIC   (c) FPGA NIC
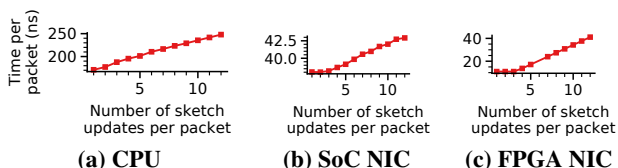
Figure 3: Phase 1 — Hashing micro-benchmark. (a) single core running OVS and sketching module. (b) 54 micro-engines. (c) single hash instance. Y axes show time per packet (ns).

number generation), and packet forwarding. For simplicity, we count the update to each row of a sketch as a separate *sketch update*, e.g., two Count-Min sketches with 4 rows each would make up 8 sketch updates per packet.

The Profiler studies the variation of time per packet as the configuration of a Count-Min sketch is varied. For hashing micro-benchmark (Figure 3), we vary the number of sketch updates for a small fixed amount of memory (to ensure memory does not become bottleneck). For the memory micro-benchmark (Figure 4a), we vary the sketch memory for a fixed number of sketch updates. We handle coin tosses similar to hashes but with sketches such as [13, 52] which rely on random number generation. Note, for all the micro-benchmarks, we generate flow keys that are uniformly distributed. This results in uniform memory access pattern for Count-Min sketch and allows us to estimate the worst case performance.

Phase 2: Composition. Given time for primitive operations from Phase 1, Phase 2 determines how different micro-benchmarks should be composed to obtain the total time per packet. The goal is to capture unique properties of how a device architecture combines the primitive operations by testing for three key properties: (1) Memory and compute concurrency, (2) Forwarding and sketching concurrency, and (3) Sketch access frequency. For each property, the Profiler defines a set of expected behaviors (composition functions). These correspond to different device architecture or sketch implementation choices. We currently rely on manually inspecting the micro-benchmarks to identify the device/implementation behavior. This process can be automated in a straightforward fashion. We detail the properties and behaviors below.

Memory and compute concurrency: Compute and memory operations in a system may be *coupled* or *decoupled* depending on the hardware: (1) in a *coupled* system, hashing and memory operations might contend for the same hardware units, (2) in the *decoupled* case, the memory and hash operations have zero contention, i.e.,

$$n_{sketch} = n_{compute} + n_{memory} \rhd \text{coupled}$$
$$n_{sketch} = \max(n_{compute}, n_{memory}) \rhd \text{decoupled}$$
$$n_{compute} = k_1 + u_h \cdot h$$
$$n_{memory} = k_2 + u_m \cdot T(m)$$

where n_{sketch} is the time for sketch updates, u_h is the number of hash computations per packet and u_m is the number of accesses to the sketch memory per packet. h is the time per hash computation, k_1 and k_2 are constants, and $T(m)$ is the time per memory access given m amount of total memory has been allocated for sketching. For the coupled case, the memory access benchmark would subsume the time for hashing and vice versa. In this case, $T(.)$ is used to represent *additional* time per memory access incurred due to potential cache misses. This is extracted by adjusting for the time per hash.

For CPU, we use the coupled model as hash computation has memory instructions which prohibit full overlap with sketch memory accesses. For the FPGA and SoC SmartNIC, we use the decoupled model, as their compute (hashing) units and memory units are completely disjoint.

For sketches which have multiple levels or control paths (e.g., UnivMon [11]), the number of primitive operations can be different for different packets based on their flow key as well as the sketch memory access pattern can be non-uniform even for uniformly distributed flow keys. In this case we interpret u_h, u_m as the expected number of operations per packet and use T(Effective uniformly accessed memory) instead of T(Total memory). For brevity we discuss the details of computing "effective uniformly accessed memory" in Appendix B. We find that UnivMon behaves as if at most 4 of its levels are accessed uniformly irrespective of the amount of sketch memory and across devices.

Forwarding and sketching concurrency: Packet forwarding and sketching can be done in parallel or in the same thread(s). If done in parallel, the time per packet would be the maximum of the inverse throughput of forwarding and sketching; otherwise, the sketching benchmarks would subsume the time for forwarding, i.e.,

$$N = \max(n_{fwd}, n_{sketch}) \rhd \text{concurrent}$$
$$N = n_{sketch} \rhd \text{sequential forwarding and sketching}$$

where N is the time per packet and n_{fwd} corresponds to the forwarding inverse throughput. For both SoC SmartNIC and CPU, sketching is done on the critical path (sequential). On the FPGA NIC sketching is off the critical path (concurrent).

Sketch access frequency: Not all sketches may be updated for every packet. For instance, the user may want certain sketches to only monitor a subset of packets forwarded by a device (users specify this by providing a *flow filter* for each sketch). This is incorporated by summing u_h, u_m weighted by the probability that a sketch is updated for a particular packet. This probability is calculated based on the flow filter and traffic matrix to obtain the fraction of packets which satisfy

the flow filter. If this cannot be computed due to granularity of traffic matrix, we keep one additional counter per sketch that counts the number of packets that update the sketch.

Phase 3: Device Configuration. The Profiler must also build a model for how sketching and forwarding performance scales with device configuration (e.g. CPU cores, micro-engines). Since performance scaling may differ across bottlenecks, we study three sketch manifests: (1) Small sketches, which trigger compute bottlenecks; (2) Single large sketch, which trigger memory bottlenecks; and (3) No sketch, to study forwarding bottlenecks. Figure 4b shows these measurements for software switch, SoC smartNIC and FPGA NIC.

Based on these measurements, the Profiler estimates f, the fraction of parallelizable execution time by fitting Amdahl's law, and updates each of n_{fwd}, $n_{compute}$ and n_{memory} to include the effect of parallelism. For instance $n_{compute}$ (from Phase 1) becomes:

$$n_{compute} = (k_1 + u_h \cdot h)[(1 - f_c) + \frac{f_c}{c}]$$

where f_c is fraction of parallelizable execution time when compute is bottleneck. We find that $f \approx 1$ for the software switch when any of forwarding, compute or memory is the bottleneck. For the SoC NIC, we find that f is ≈ 1 when forwarding or hashing is the bottleneck. However, when memory is the bottleneck, increasing micro-engines does not change packet rate, implying that f is 0 (Figure 4b). This is consistent with the fact that there is a single cache and DRAM (where sketch memory is allocated) – shared by all micro-engines – which becomes a bottleneck, as opposed to cores on a CPU which have their own caches, which allows for parallelism even when memory is the bottleneck. For the FPGA NIC, f is ≈ 1 when hashing is bottleneck otherwise it is 0. We measure the compute resources of FPGA in the units of a hash unit instance, each consuming 5 digital signal processors, 214 lookup tables, and 486 flip-flops.

Summary. The final relations encoding time per packet N in terms of number of operations u_h, u_m, sketch memory m and device parameters (h, $T(.)$, f's, constants) and device configuration are referred to as *device profiles*. This algebraic characterization of performance-resource trade-offs is used by the Optimizer for deciding sketch placement and resource allocation. In our current formulation, we don't explicitly model the impact of contention from non-monitoring tasks. We assume that compute resources are pinned to sketches and memory resources are explicitly allocated using technologies akin to Intel Cache Allocation Technology [54]. We also don't study the overhead of such isolation mechanisms.

Programmable switch [16] is a special case here as its resources are allocated by the compiler with guaranteed constant time per packet. Thus, the Profiler only needs to model the resources for different sketching manifests based on the resource usage output of the compiler.
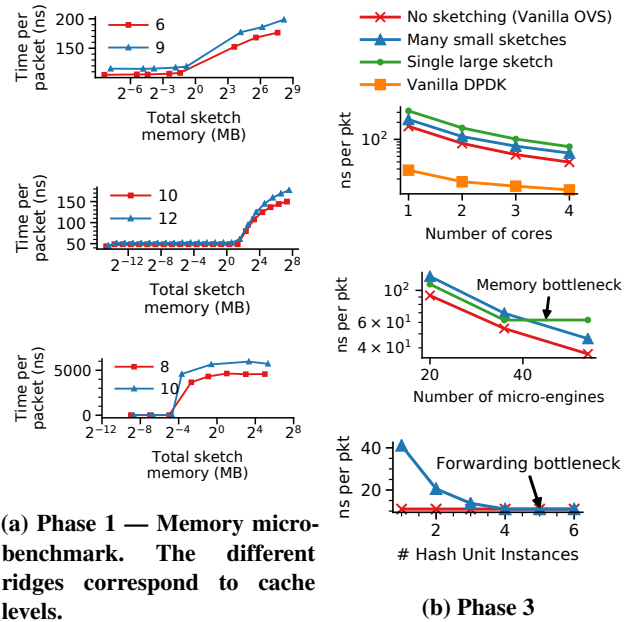
(a) **Phase 1 — Memory microbenchmark. The different ridges correspond to cache levels.**

(b) **Phase 3**

Figure 4: (a) Phase 1 — CPU with single core running OVS and sketching module (top), SoC NIC with 54 micro-engines (middle), FPGA with 10 hash instances (bottom). The numbers in the legend correspond to the number of sketch updates per packet.

(b) Phase 3 — Inverse throughput as CPU cores (top), SoC NIC micro-engines (middle) and FPGA hash unit instances (bottom) are varied. For FPGA NIC, single large sketch (not shown due to scale) is a flat line (memory bottleneck).

5 Optimizer

The goal of the Optimizer is to decide which sketch should be placed on which device and which resources each device should use while meeting the device constraints, monitoring requirements, traffic demands, and optimizing towards user-specified goals. We formulate the placement and resource allocation problem as a Mixed-Integer Bi-linear program (MI-BLP), which is defined below with constants and variables described in Tables 3 and 4 respectively. While we investigate resource usage as an objective for concreteness, our formulation can be easily tweaked to handle other objective functions (Equation 7 in Appendix C).

Input: The input has the following three key features: (1) set of *devices* $\mathcal{D}$ in the network, along with their profiles generated using the Profiler (§4) and resource availability; (2) set of needed *sketches* $\mathcal{S}$, along with their configuration; (3) set of *Origin Destination (OD) pairs* $\mathcal{P}$.

In particular, each OD-pair is uniquely specified by: (1) device-level path in the network, (2) rate of traffic demand on that path, (3) set of sketches that should monitor traffic that is part of this OD-pair. With the OD-pair abstraction, we can handle the following cases:

- If there are multiple paths between an OD-pair, logically

$$\text{O1: resources} \quad \text{Minimize} \sum_{d \in \mathcal{D}} (res_d + mem_d), \quad \textbf{s.t.} \quad (1)$$

$$\text{C1: coverage} \quad \sum_{d \in p_\pi} b_{(d,s)} \geq 1 \quad \forall p \in \mathcal{P}, \forall s \in p_s$$

$$\text{C2: accuracy} \quad mem_{(d,s)} \geq s_{mem} \cdot b_{(d,s)} \quad \forall s \in \mathcal{S}, \forall d \in \mathcal{D}$$

$$\text{C3: capacity} \quad \sum_{s \in \mathcal{S}} b_{(d,s)} \cdot s_{rows} \leq d_{rows}, \quad \text{and}$$

$$mem_d = \sum_{s \in \mathcal{S}} mem_{(d,s)} \leq d_{mem} \quad \forall d \in \mathcal{D}$$

$$\text{C4: profiles} \quad \forall d \in \mathcal{D}:$$
$$time_d = d_{time}(res_d, \mathcal{P}_d, \{(mem_{(d,s)}, b_{(d,s)}) | s \in \mathcal{S}\})$$

$$\text{C5: traffic} \quad time_d \leq \frac{1}{d_{\text{traffic}}} \quad \forall d \in \mathcal{D}, \quad \text{where}$$
$$d_{\text{traffic}} = \sum_{p \in \mathcal{P}_d} p_t, \quad \mathcal{P}_d = \{p | d \in p_\pi, p \in \mathcal{P}\}$$

Symbol	Interpretation
$\mathcal{D}, \mathcal{S}, \mathcal{P}$	Set of devices, sketches, and OD-pairs
p_s	Set of sketches for OD-pair p
p_π	Device level path for OD-pair p
p_t	Rate of traffic relevant for OD-pair p
s_{mem}, s_{rows}	Memory required for desired accuracy & number of rows for sketch s
d_{mem}	Maximum memory on device d
d_{rows}	Maximum rows that can fit on device d
d_{time}	(Function) representing the device profile (§4) for d (Time per packet in seconds)
$\mathcal{P}_d$	OD-pairs which pass through d
$d_{traffic}$	Total traffic rate in packet per second (pps) witness by device d

Table 3: Constants.

Domain	Symbol	Interpretation
$\mathbb{R}_{\geq 0}$	$mem_{(d,s)}$	Memory of sketch s on device d.
$\{0,1\}$	$b_{(d,s)}$	Is sketch s placed on device d
$\mathbb{Z}_{\geq 0}$	res_d	Resources allocated to device d, (e.g., processing cores, stages, functional units etc.)

Table 4: Variables.

distinct OD-pairs can be instantiated for each separate path.

- If part of the traffic in an OD-pair needs to be monitored by sketch A and other part using sketch B, then two logically distinct OD-pairs can be instantiated with the same path but specifying different sketches along with the appropriate rate of traffic which is relevant to each sketch.

- If the same query or sketch needs information about traffic on multiple OD-pairs, each OD-pair can refer to the same sketch identifier.

- Multiple sketches can monitor traffic in a single OD-pair. This would be used in the cases when we need to maintain a statistic for different dimensions of the same traffic (e.g., dimension 1: distribution of DstIPs for each source and dimension 2: distribution of SrcIPs for each destination).

This input is compiled from the high-level measurement requirements specified by the user. The traffic demands (packet rates) are estimated using the traffic matrix and the paths are obtained using the routing information and flow filters specified for each sketch by the user.

Outputs and constraints: The Optimizer decides which sketch should be placed on which device. This is indicated through variables $b_{(d,s)}$. While doing so, the Optimizer ensures that for each OD-pair, each sketch of that OD-pair is placed on at least one of the devices lying on the OD-pair's path (C1: flow coverage in Equation 1). The memory for each sketch is directly determined by the accuracy required for that sketch (C2: monitoring accuracy). Each device is constrained by memory capacity and some devices may have constraints on row capacity (e.g., due to limited stages in programmable switch) (C3: device capacity). C4 (device profiles) encodes the relationship between time per packet, the sketch parameters, device parameters and the device configuration as described in Section 4. The processing overhead on each device should be such that the overhead does not stall the traffic flowing through the device (C5: forwarding performance). Note that C4 is natively expressed through Gurobi's API using piece-wise linear [55] and bi-linear constraints [56]. We elaborate on this in Appendix C.

Measurement Accuracy: Different feasible solutions may deploy sketches at different locations in a network (e.g., Figure 1) and even create multiple instances of the same sketch. We note that our sketch placements make no impact on the monitoring accuracy. This is because, these multiple instances are linearly merged (§2) at the central controller. The merge is possible as instances of the same sketch share the same hash functions, memory configuration. The merge does not lead to over/under counting as we ensure that each packet updates exactly one instance of all the required sketches. This is ensured as: (1) constraint C1 guarantees that there is at least one instance of the required sketches on the path of each OD-pair, and (2) we generate sketch manifests so that exactly one of these instances is chosen (arbitrarily) to be updated for each OD-pair.

6 Scalability and Dynamics

Solving the MI-BLP in Gurobi can take more than a few hours even for modestly sized data center topologies with thousands of devices. For quick responses to network dynamics and scaling to more devices, we use a three step approach:

- *Step 1*: Partition the network topology into disjoint clusters
- *Step 2*: Run Optimizer to assign sketches to the clusters[2]
- *Step 3*: For each cluster, run Optimizer to place sketches onto devices *within* the cluster.

Since the placement decision for each cluster is done independently, Optimizer instances can be spawned in parallel,

---

[2]For this step, the traffic demands and device profiles (C4-5 in equation 1) are not used, as the Profiler does not model the performance of clusters.
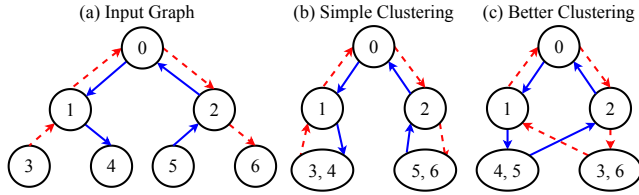
Figure 5: Different ways to cluster a graph.

which makes this approach scalable. Note that, Step 2 itself can consume significant time if the number of clusters is large. We address this by recursively applying Step 1 and 2 to build a hierarchy of clusters. We determine the threshold to apply the recursive step by modeling the cluster size-compute time relationship (Figure 14 in Appendix A) and choosing the largest cluster size with an acceptable run-time. We also add a *fast path* for quick response to cater to network dynamics that *directly affect* only a small subset of devices.

6.1 Clustering Approach

Partitioning the network devices into clusters will inevitably hide some details about the topology and create a trade-off between optimality and solving time. This is because when the Optimizer is run to place sketches onto the cluster (Step 2), it may choose a sub-optimal (or even infeasible) cluster as it does not know what is inside the cluster. Ideally, we want to cluster the topology in a way that significantly accelerates the solving process while incurring minimal optimality loss.

Clustering Examples. We observe that naïvely clustering the topology graph based on the hierarchy of the topology or applying graph clustering algorithms (e.g., spectral clustering [51]) can lead to sub-optimal or even infeasible sub-problems. Figure 5 illustrates this in an example topology of 7 devices. In this example, the edges show the paths of the OD-pairs and the colors (or line-styles) of the edges correspond to different OD-pairs. We need to deploy 2 sketches, each of which monitors one of the OD-pairs shown in blue (solid) and red (dotted).

Simple clustering: By directly applying spectral clustering on network topology, we obtain the result in Figure 5(b). Based on this clustering, assume that in *Step 2*, the Optimizer decided to place one sketch on cluster (3,4) and the other sketch on cluster (5,6). Further assume that the sketch placed on cluster (3,4) monitored the red OD-pair. When the Optimizer again runs *Step 3* for devices within cluster (3,4), since only device 3 sees packets on the red path, the sketch for the red OD-pair can only be placed on device 3. If device 3 is currently not available to place the sketch or device 6 is in fact a better allocation, the simple clustering will lead to an infeasible or sub-optimal solution.

Better clustering: If we cluster as Figure 5(c), the sketch for the red (dotted) OD-pair could be assigned to cluster (3,6) and the Optimizer running within that cluster retains its freedom to place the sketch on device 3 or 6.

Our Design. We learned from the above example that we should keep nodes that *communicate* with each other in the same cluster, where communicate means that there is an OD-pair that has a path connecting the nodes. This should be done irrespective of the number of network-level hops (physical or logical links) between them. This provides the Optimizer in *Step 3* additional placement choices for sketches within the cluster sub-problem.

We can incorporate communication affinity by applying spectral clustering on the communication graph (G_c), where vertices are network devices, with edges between all pairs of devices which communicate with each other. While this approach works for general network environments, we find that spectral clustering itself is time consuming for large networks [57]. Thus, we imitate spectral clustering using a domain-specific heuristic and are investigating faster alternatives and implementations of spectral clustering.

Our heuristic is based on the observation that many clustering solutions preserve enough flexibility for sketch placement yielding good performance (Figure 12 in Appendix A). This observation was made when exploring the space of possible clustering solutions and their impact on MIP objective using simulated annealing [58]. Our heuristic works in a multi-tenant setting where tenants share the network but not the end-hosts. For building clusters, we instantiate a cluster for end-hosts of each tenant. Then, to ensure clusters are evenly sized, we arbitrarily merge (or split) the clusters if they are too small (or big). Finally, the switches and NICs are assigned to the cluster of the end host with which they have highest affinity. In Appendix A, we discuss how to choose cluster sizes and provide examples of clustering output for different clustering techniques.

6.2 Fast Path

The Fast Path further improves response time for network dynamics including: (1) Topology change in the path (e.g., VM migration); (2) Monitoring query (sketch) change from the user/operator; (3) Traffic change in the OD-pair (e.g., traffic demand variations); (4) Available resource change due to the dynamics of other non-monitoring services running on the shared devices. (1)–(3) reflect as changes in OD-pairs ($\mathcal{P}$) and (4) reflects as changes in devices ($\mathcal{D}$).

The key observation we have from §6.1 is that sub-problems should preserve enough placement choices for sketches. Based on that, we find that, on network change events, recomputing placement only for the set of devices $\mathcal{A}$ *directly affected* by the changes is sufficient. We compute this set through the following process: (a) For each changed device $d \in \mathcal{D}$, we add d to $\mathcal{A}$ and, for each sketch s currently placed on d, we add the OD-pair(s) monitored by s to the set of changed OD-pairs. (b) For each changed OD-pair (due to previous step or otherwise), we add all devices specified in the path of the OD-pair to $\mathcal{A}$. The Optimizer is then run

only for the devices in set $\mathcal{A}$ and the sketches specified in the changed OD-pairs, including the sketches already mapped to devices in $\mathcal{A}$.

7 Evaluation and Implementation

We implement HeteroSketch and evaluate its effectiveness. Our major findings are as follows:

- HeteroSketch's Profiler accurately characterizes the performance of devices across a variety of sketch manifests and device configurations (§7.1)
- HeteroSketch's Optimizer is able to (1) reduce resource footprints and obtain feasible solutions when prior approaches fail, and (2) scale to large topologies (> 40,000 devices) while preserving good quality solutions. (§7.2)
- HeteroSketch's Fast Path allows prompt responses to network dynamics including changes in topology, traffic, query, and available resources. (§7.3)

Implementation. For the software switch, the sketching modules are implemented as a part of the OVS data plane. For the SmartNIC, we use the internal and external memory regions for storing sketch state as these are accessible from all the micro-engines unlike local and island-specific memory regions. The Optimizer is run on an Intel® Xeon® CPU E5-2680 v2 processor @ 2.80GHz with 128 GB RAM.

7.1 Performance Profiler

Setup. This evaluation uses the same three device setup introduced in §4 which was used to create the device profiles (Table 2). We use dpdk-pktgen [59] to generate traffic and configure dpdk-testpmd [60] to measure receive rate. The sender generates min-sized (64 Byte) packets to measure the maximum packets per second that can be processed. We use source IPs as the flow keys for the sketches, which are taken from a uniform random distribution. This is done in order to use the Optimizer to allocate resources for worst-case (uniform) traffic scenarios. We discuss in Appendix B, how other traffic distributions could be accommodated.

Workloads. For generating sketch manifests, we consider a range of configurations for three different types of sketches: Count-Min, Count Sketch and UnivMon. For each sketch, we vary the number of rows from 1 to 12 and, for memory, we vary the counters per row from 1 to 2^{22} (≈ 4 million) in steps of powers of 2. For UnivMon, we vary levels from 2^2 to 2^5 in steps of powers of 2. For generating device configurations, we vary the SoC NIC micro-engines from 20 to 54, for the software switch, vary cores from 1 to 4, and for the FPGA NIC, vary maximum allowed hash instances from 1 to 12. Large number of rows emulate multiple sketches per device.

Results. Table 5 summarizes the results of the experiments and Figure 6 shows results for a subset of the experiments.

Sketch	CPU	SoC NIC	FPGA
Count-Min Sketch	3.08, 9.63	3.68, 12.99	1.9, 4.1
Count Sketch	5.61, 8.51	1.26, 4.51	2.2, 4.14
UnivMon	2.80, 6.38	2.38, 3.75	2.28, 5.88

Table 5: Profiler Evaluation — Each sketch–device combination reports the (mean, 90th percentile) of percent error $\left(\left|\frac{actual-model}{actual}\right| * 100\right)$ for the time per packet metric.

(a) 2 Cores   (b) 4 Cores
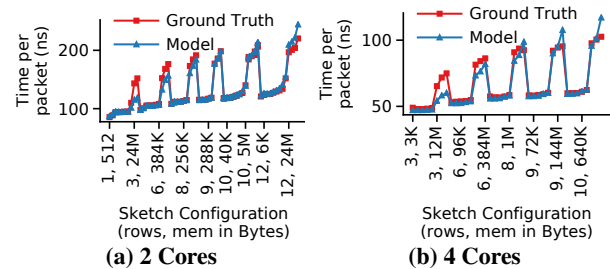
Figure 6: Performance Profiler — CPU model evaluation for Count-Min Sketch.

Figures for additional experiments can be found in Appendix B. These figures compare the time per packet estimated by the Profiler's model and the ground truth. Over all the combinations of sketch manifests and device configurations, the Profiler's model is within 5.61% of the ground truth on average and within 13% in the tail cases. We don't show results for programmable switch as it guarantees line rate if the program fits (which is captured by the capacity constraints in the Optimizer). We observe that most profiling errors occur in larger sketch configurations, when off-chip memory is used. Based on our estimates, such errors would disturb the resource allocation for less than 5% devices when profiles are off by 10%. We discuss the impact of profiling errors on the Optimizer in more detail in Appendix B.

7.2 Optimizer

We evaluate the solutions generated by the Optimizer on two key metrics: (1) resource-efficiency benefits and (2) the optimization run time. We use the device profiles to estimate the resource usage and performance for the sketch placements generated by different optimization schemes. We use the following methodology to generate input scenarios:

Topologies. We conduct two studies: (a) *Topology* – variation with topology structure. We use selection of topologies from the Internet Topology Zoo [61], JellyFish [62] and a three-level Fat-Tree data center topology (Clos) [63, 64] (Figure 7a). (b) *Scale* – variation with topology size. We use a Clos topology with varied degrees (number of *pods*) from 16 to 48 (Table 6, Figure 7b). We extend the Points-of-Presence (POP) topologies [61] to include servers and NICs based on principles from [65].

For both studies, we focus on a multi-tenant setting, where different tenants submit monitoring tasks to a central system (e.g., the cloud or Internet service provider). This allows us
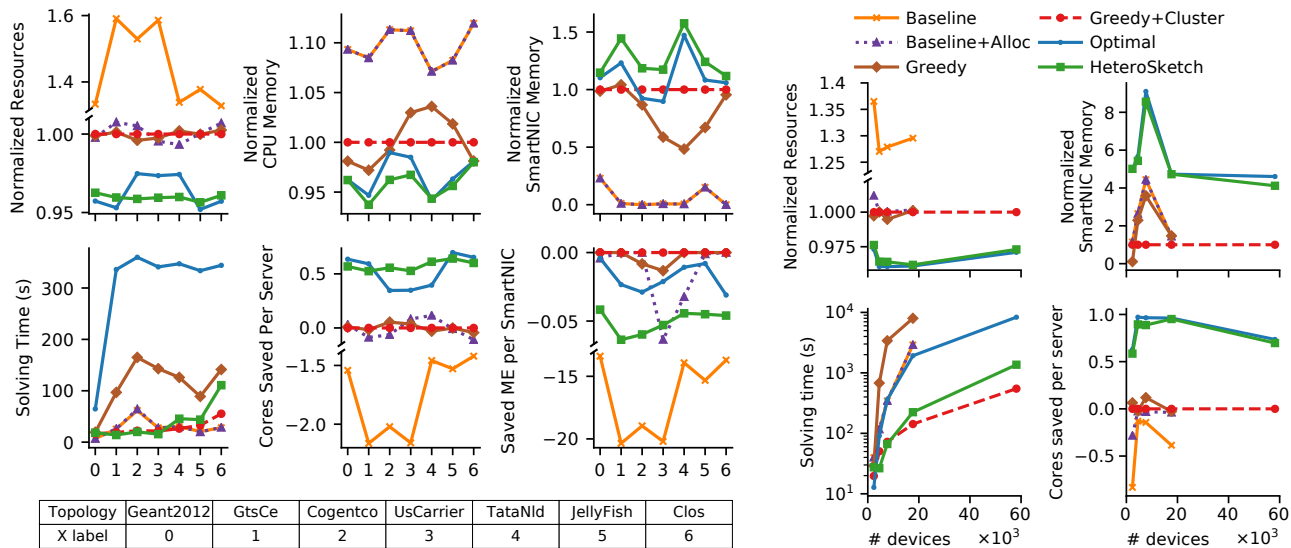
(a) **Topology Study** — The Clos topology used has 20 pods, JellyFish has the same number of devices as the Clos topology (2000 servers/NICs each and 500 switches). Note, a time limit of 300s for the MIP solver was used.

(b) **Scale Study** — These experiments were performed on inputs described in Table 6.

Figure 7: Optimizer Evaluation — Compute resources are shown in terms of amount saved relative to Greedy+Cluster (negative compute resources implies more resource consumption than Greedy+Cluster). Total resources and memory resources are normalized w.r.t Greedy+Cluster. Both sub-figures share the same legend. Additional details in Appendix C (Figures 20 & 21).

Fat-tree Degree	Sketch load (Y)	Servers / NICs	Switches	Total devices
16	1	1,024	320	2,368
20	3	2,000	500	4,500
24	3	3,456	720	7,632
32	4	8,192	1,280	17,664
48	4	27,648	2,880	58,176

Table 6: Topologies and Workloads.

to stress test the schemes under a diverse set of monitoring requirements. We assign X servers to each tenant where X is taken from the uniform distribution $\mathcal{U}(6, 12)$. We set the capacity of each server link to 25 Mpps (64B packets). This is equivalent to the throughput of vanilla OVS with 4 cores. We randomly assign half of the servers with the SoC SmartNIC and other half with the FPGA NIC.

Monitoring requirements. The total number of queries (sketches) is set equal to Y times the number of servers in the network. These monitoring tasks are evenly partitioned among the tenants. We vary the sketching load (Y) between 1 (low) and 4 (high). Low load is used to study the system when each device in the network runs much below the total monitoring capacity they can handle. High load is used to study the system under stress. Table 6 shows the load used for different topologies. The monitoring tasks are equally divided between Count-Min, Count Sketch and UnivMon sketches.

OD-pairs. Each tenant specifies M OD-pairs from the set of servers assigned to them, where $M \sim \mathcal{U}(64, 96)$ and each OD-pair is monitored by K randomly chosen sketches of the tenant where $K \sim \mathcal{U}(1, 3)$. Since we don't have access to the monitoring demands from different operators, we select

OD-pairs, routes (paths) and traffic demands iteratively to ensure: (1) traffic is evenly distributed between OD-pairs, and (2) link utilization is at least 90% to stress the system.

Compared Schemes. As shown in Figure 7b, we compare HeteroSketch (6) against five other schemes (1)–(5):

(1) Baseline: Capacity-aware placement with *static* resource allocation, i.e., placing sketches to minimize the sketch memory with compute resource assigned apriori to cores=5, micro-engines=54 (equal to resources exposed to Optimizer). This is closest to UnivMon [11].

(2) Baseline+Alloc: The placement of sketches is done in the same manner as in Baseline. Instead of static resource assignment, just enough resources are allocated to meet the traffic and sketching demands based on the device profiles. This is used to investigate benefits obtained solely from profiling-aware resource allocation.

(3) Greedy: This is a strawman extension to the baseline which prioritizes placing sketches on programmable switches over CPUs and SmartNICs because of their line-rate guarantees. Resource allocation is done using device profiles similar to Baseline+Alloc. Prioritizing sketch deployment on switches is a reasonable heuristic when sketch load (Y) is low (first data point of Figure 7b, gap between resource usage for Baseline+Alloc and Greedy).

(4) Greedy+Cluster: To compare the optimality of our scheme to prior work for larger topologies, we extend the Greedy strategy to use our clustering optimization.

(5) HeteroSketch w/o clustering (Optimal): Joint placement and resource allocation using the formulation in Equation 1

Sketch	CPU	SoC NIC	FPGA	Switch
Count-Min	4112	2001	2111	1563
Count	4172	2066	2106	232
UnivMon	4169	2120	2049	143

Table 7: Sources of benefits (Sketch-Device Affinity) — Total number of sketches on each device. (Clos $pods = 16$, $Y = 4$.)

without the clustering approach.

(6) HeteroSketch: Joint placement and resource allocation with the clustering optimization.

Results. In Figure 7, we can see that HeteroSketch is able to lower resource utilization significantly (close to optimal) and saves between half to one CPU core per server on average compared to Greedy+Cluster ($20-30\%$ improvement). Compared to Baseline, HeteroSketch saves around 2.5 cores per server and about $15-20$ micro-engines per SoC NIC ($40-60\%$ improvement). It accomplishes this without incurring significant time to compute the placement of measurement tasks even as topologies scale to more than 40k devices. We compute total resources (HeteroSketch MIP objective) as a weighted sum of all devices resources including CPU cores, micro-engines, FPGA hash unit instances and memory, and normalize it by the total resources used by Greedy+Cluster. This is shown as *normalized resources* in Figure 7.

Sources of Benefits. We explore the benefits obtained from different features of HeteroSketch.

Bottleneck awareness: The device profiles are successful in incorporating capabilities of different device architectures, this allows HeteroSketch to allocate just enough resources to meet the sketching and forwarding demands. The gap in normalized resources between Baseline and Baseline+Alloc in Figures 7a and 7b demonstrates benefits attained solely from profiling aware resource allocation. For instance, on the SoC SmartNIC, when memory is the bottleneck, more micro-engines do not improve forwarding performance (Figure 4b).

Efficient use of resources: We see from Figure 8a that HeteroSketch allocates 20% fewer CPU cores (8k vs 10k cores) but uses the cores it allocates more effectively (each core is >80% utilized). Specifically, on the software switch, the cores are configured to poll NICs for packets (for performance reasons), as a result, CPU cycles are wasted busy polling when there are no packets in the NIC buffers. With the help of device profiles, the Optimizer is able to consolidate load towards cores which would otherwise waste cycles. Similar trends are observed for other resources including SoC memory bandwidth (Figure 8b), and SoC micro-engines (Figure 8c).

Ability to trade-off resources: For the scale study (Figure 7b), we set lower weightage to SoC SmartNIC memory relative to the topology study (Figure 7a). We observe in Figure 7b that HeteroSketch is able to incorporate this by saving more number of cores per server at the cost of SoC SmartNIC memory usage.

Sketch-Device affinity: We show the number of sketches instantiated of each type on each device in Table 7. Recall that

(a) CPU Utilization.

(b) SoC SmartNIC Memory Bandwidth Utilization.

(c) SoC SmartNIC Micro-engine Utilization.
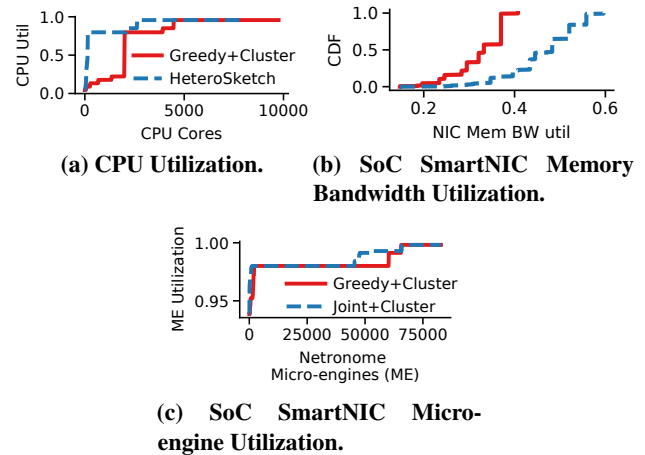
Figure 8: Sources of Benefits (Efficient use of resources) — (Clos $pods = 20$).

the monitoring requirement specified equal number (≈ 1360) of queries for each sketch type. Multiple instances of each sketch are created to meet the coverage requirements. We make three key observations here: (1) HeteroSketch tries to place heavier sketches (e.g. UnivMon) on better vantage points so as to reduce the total required instances. (2) The switch statically allocates resources to each sketch while other devices can share resources across sketches. Due to this HeteroSketch places less number of sketches on the switch, especially for Count Sketch and UnivMon due to more number of operations. (3) HeteroSketch instantiates relatively more number of UnivMon sketches on CPU and SoC SmartNIC as they have relatively larger and faster memories.

Clustering Algorithm. Figure 7b and 7a also suggest that our clustering technique does not significantly degrade resource efficiency. Clustering imposes a trade-off between optimality and solving time which we explore in more detail in Appendix A. We see in Figure 7a, that HeteroSketch finds better solutions than the Optimal scheme when configured with a time limit, achieving a better trade-off between optimality and solving time than the MIP solver.

For evaluating the Optimizer, we used the multi-tenant clustering heuristic developed in §6.1. We investigated use of other algorithms to cluster the communication graph (§6.1) including KMedoids, HDBSCAN, modularity maximization [57, 66]. Unfortunately, these techniques yield infeasible sub-problems in Step 3 of §6.1 for the inputs that we used.

Note, while it is true that our assumption about one server being solely used by one tenant makes the optimization problem instance easier, despite that, the MIP solver still needs explicit guidance in the form of clustering for speed up. This can be seen from difference in solving time with and without clustering in Figure 7.
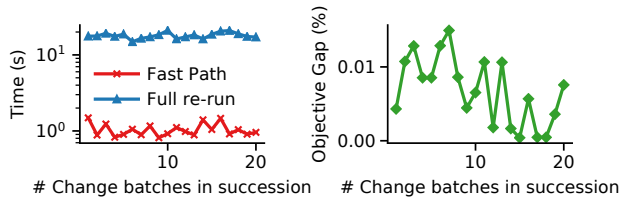
Figure 9: Dynamics — variation with number of changes in succession (Clos topology with 16 pods).
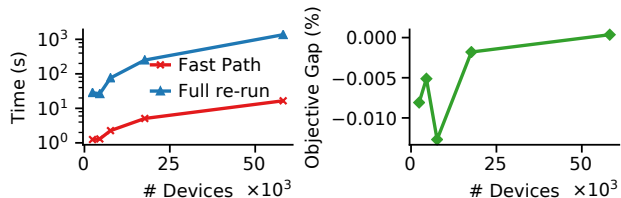
Figure 10: Dynamics — variation with topology size for a single change. Negative objective gap means that Fast Path consumes less resources than full re-run. This can happen because both use the clustering heuristic which does not guarantee strictly optimal solutions. (Conducted on topologies in Table 6.)

7.3 Dynamics - Fast Path

As described in §6, changes in OD-pairs can correspond to changes in traffic, monitoring requirement, and topology. To generate such changes, we generate inputs for the Optimizer as described in §7.2. Then, we randomly keep $10n$ of the generated OD-pairs aside to serve as change batches, where n is the number of batches and each batch has 10 changed OD-pairs. We also generate resource availability changes in each batch by randomly sampling 10 devices and randomly increasing/decreasing their resource availability (e.g., micro-engines/memory) by 20%. We find that changes in a batch amount to roughly 50 to 150 devices being directly affected ($|\mathcal{A}|$ in §6.2) across the topologies in Table 6. In general, $|\mathcal{A}|$ can depend on the size of the topology and monitoring requirements. For the Clos topology, since path lengths are equal irrespective of number of pods, we see little dependence of pod count on $|\mathcal{A}|$.

Despite being run only for the subset of affected devices, successive runs of the Optimizer don't diverge from the global solution generated by re-running the Optimizer over the entire topology using clustering (Figure 9). Further, the response time to cater to dynamics is reasonably low with the Fast Path even as we scale the topology size (Figure 10). The Fast Path uses the clustering optimization when the set of affected devices is larger than the optimal clustering size and can amount to a full run of the Optimizer in the worst case. Fast Path, together with the clustering heuristic enables a response time of a few seconds (the set of affected devices is small) to a few minutes (when run over entire network). Note, within the scope of this work, we don't study how the new placement can be configured consistently and quickly. These are active areas of research [35, 67].

8 Other Related Work

Other related work not covered in §2 can be classified into four categories:

Sketch resource allocation. SCREAM [68] allocates memory for a sketch based on temporal and spatial changes in traffic moments for a fixed sketch placement. Open Sketch [10] optimally selects sketch algorithm and configuration for a given query. Both are complementary to our work.

Sketch implementations for different hardware. Our work relies on state-of-the-art implementations of sketches for different hardware. Fortunately, recent efforts [12, 13, 22] have focused on addressing the bottlenecks of sketching algorithms in software switches and have demonstrated efficient implementations in programmable switches or NICs [11, 12, 31].

Other work in network monitoring. HeteroSketch's goal is to support network-wide flow monitoring. Numerous complementary efforts focus on either fine-grained and adaptive flow monitoring (e.g., [36, 38, 69]), diagnosis (e.g., [70–72]), or network performance-related objectives (e.g., [73, 74]).

Efforts in speeding up network-wide optimizations. Concurrent with our work, Abuzaid et al. in [75] explore the use of clustering to speed up network flow problems. While our work tries to maintain feasiblity of sub-problems through carefully deciding which devices to cluster together, they impose additional constraints while conducting flow allocations over clusters to ensure that the subsequent sub-problems are feasible. We leave exploration of such a technique in the context of sketch placement for future work.

9 Conclusions

We observe that existing efforts on sketch-based network-wide monitoring remain impractical as they fail to cope with the key requirements of heterogeneity and dynamics in the network. We propose HeteroSketch as a coordinated solution to achieve optimized task placement and resource allocation over heterogeneous networks. HeteroSketch precisely characterizes the performance of sketches on diverse devices and is integrated with a clustering technique to handle networks scale and dynamics. Our evaluation demonstrates that HeteroSketch scales to toplogies with tens of thousands devices with near optimal resource efficiency. We posit that our system can more generally be applied to allocate resources for other networked applications in heterogeneous networks, and we plan to explore this for future work.

References

[1] Mohammad Alizadeh et al. "CONGA: Distributed congestion-aware load balancing for datacenters". In: *Proceedings of the 2014 ACM conference on SIG-COMM*. 2014, pp. 503–514.

[2] Mohammad Alizadeh et al. "PFabric: Minimal near-Optimal Datacenter Transport". In: *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*. SIGCOMM '13. Hong Kong, China: Association for Computing Machinery, 2013, pp. 435–446.

[3] Theophilus Benson et al. "MicroTE: Fine grained traffic engineering for data centers". In: *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*. 2011, pp. 1–12.

[4] James McCauley et al. "Thoughts on Load Distribution and the Role of Programmable Switches". In: *ACM SIGCOMM Computer Communication Review* 49.1 (2019), pp. 18–23.

[5] Rui Miao et al. "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 2017, pp. 15–28.

[6] Pedro Garcia-Teodoro et al. "Anomaly-Based Network Intrusion Detection: Techniques, Systems and Challenges". In: *computers & security* 28.1-2 (2009), pp. 18–28.

[7] L. Ying, R. Srikant, and X. Kang. "The power of slightly more than one sample in randomized load balancing". In: *2015 IEEE Conference on Computer Communications (INFOCOM)*. 2015, pp. 1131–1139.

[8] Graham Cormode. "Count-Min Sketch". In: *Encyclopedia of Algorithms*. Ed. by Ming-Yang Kao. Boston, MA: Springer US, 2008, pp. 1–6.

[9] Moses Charikar, Kevin Chen, and Martin Farach-Colton. "Finding frequent items in data streams". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 2380 LNCS. 2002, pp. 693–703.

[10] Minlan Yu, Lavanya Jose, and Rui Miao. "Software Defined Traffic Measurement with OpenSketch". In: *Nsdi '13*. 2013, pp. 29–42.

[11] Zaoxing Liu et al. "One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM '16. Florianopolis, Brazil: Association for Computing Machinery, 2016, pp. 101–114.

[12] Tong Yang et al. "Elastic sketch: Adaptive and fast network-wide measurements". In: *SIGCOMM 2018 - Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 2018, pp. 561–575.

[13] Zaoxing Liu et al. "Nitrosketch: Robust and general sketch-based monitoring in software switches". In: *Proceedings of the ACM Special Interest Group on Data Communication*. 2019, pp. 334–350.

[14] B. Claise. *Cisco systems NetFlow services export version 9*. RFC 3954.

[15] Peter Phaal, Sonia Panchen, and Neil McKee. "InMon corporation's sFlow: A method for monitoring traffic in switched and routed networks". In: (2001).

[16] *Tofino 2 | Barefoot*. URL: https : / / www . barefootnetworks . com / products / brief – tofino-2 (visited on 04/28/2020).

[17] *Broadcom Trident 3*. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56870-series/.

[18] *Netronome*. URL: https : / / www . netronome . com / products / smartnic / overview (visited on 04/28/2020).

[19] Daniel Firestone et al. "Azure Accelerated Networking: SmartNICs in the Public Cloud". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, Apr. 2018, pp. 51–66.

[20] Ben Pfaff et al. "The Design and Implementation of Open vSwitch". In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, May 2015, pp. 117–130.

[21] *BESS: Berkeley Extensible Software Switch*. URL: http://span.cs.berkeley.edu/bess.html (visited on 09/12/2020).

[22] Qun Huang et al. "Sketchvisor: Robust network measurement for software packet processing". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 2017, pp. 113–126.

[23] Vyas Sekar et al. "CSAMP: A System for Network-Wide Flow Monitoring". In: *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. NSDI'08. San Francisco, California: USENIX Association, 2008, pp. 233–246.

[24] Xin Jin et al. "Netchain: Scale-free sub-rtt coordination". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 2018, pp. 35–49.

[25] Xin Jin et al. "Netcache: Balancing key-value stores with fast in-network caching". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. 2017, pp. 121–136.

[26] Rui Miao et al. "SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '17. Los Angeles, CA, USA: Association for Computing Machinery, 2017, pp. 15–28.

[27] *Alveo U280 Data Center Accelerator Card*. URL: https://www.xilinx.com/products/boards-and-kits/alveo/u280.html (visited on 03/05/2021).

[28] Antonis Manousis et al. "Contention-Aware Performance Prediction For Virtualized Network Functions". In: *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*. SIGCOMM '20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 270–282.

[29] Mihai Dobrescu, Katerina Argyraki, and Sylvia Ratnasamy. "Toward Predictable Performance in Software Packet-Processing Platforms". In: *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX Association, Apr. 2012, pp. 141–154.

[30] *Gurobi - The fastest solver - Gurobi*. URL: https://www.gurobi.com (visited on 04/29/2020).

[31] Vibhaalakshmi Sivaraman et al. "Heavy-hitter detection entirely in the data plane". In: *Proceedings of the Symposium on SDN Research*. 2017, pp. 164–176.

[32] Pat Bosshart et al. "Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN". In: *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*. SIGCOMM '13. Hong Kong, China: Association for Computing Machinery, 2013, pp. 99–110.

[33] Yin Zhang et al. "On the characteristics and origins of internet flow rates". In: *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*. 2002, pp. 309–322.

[34] Masoud Moshref et al. "Scalable Rule Management for Data Centers". In: *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX, 2013, pp. 157–170.

[35] Qun Huang et al. "OmniMon: Re-Architecting Network Telemetry with Resource Efficiency and Full Accuracy". In: *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*. SIGCOMM '20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 404–421.

[36] Arpit Gupta et al. "Sonata: Query-driven streaming network telemetry". In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 2018, pp. 357–371.

[37] Lihua Yuan, Chen-Nee Chuah, and Prasant Mohapatra. "ProgME: towards programmable network measurement". In: *IEEE/ACM Transactions on Networking* 19.1 (2010), pp. 115–128.

[38] Rob Harrison et al. "Network-wide heavy hitter detection with commodity switches". In: *Proceedings of the Symposium on SDN Research*. 2018, pp. 1–7.

[39] Masoud Moshref et al. "DREAM: Dynamic Resource Allocation for Software-Defined Measurement". In: *Proceedings of the 2014 ACM Conference on SIGCOMM*. SIGCOMM '14. Chicago, Illinois, USA: Association for Computing Machinery, 2014, pp. 419–430.

[40] Nick Duffield, Carsten Lund, and Mikkel Thorup. "Estimating flow distributions from sampled flow statistics". In: *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*. 2003, pp. 325–336.

[41] Cristian Estan and George Varghese. "New directions in traffic measurement and accounting". In: *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*. 2002, pp. 323–336.

[42] Noga Alon, Yossi Matias, and Mario Szegedy. "The space complexity of approximating the frequency moments". In: *Journal of Computer and system sciences* 58.1 (1999), pp. 137–147.

[43] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. "SpaceSaving: Efficient Computation of Frequent and Top-k Elements in Data Streams". In: *Proceedings of the 10th international conference on Database Theory*. 2005, pp. 398–412.

[44] Balachander Krishnamurthy et al. "Sketch-based change detection: methods, evaluation, and applications". In: *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*. 2003, pp. 234–247.

[45] George Nychis et al. "An empirical evaluation of entropy-based traffic anomaly detection". In: *Proceedings of the 8th ACM SIGCOMM conference on Internet measurement*. 2008, pp. 151–156.

[46] Peter Clifford and Ioana Ada Cosma. "A simple sketching algorithm for entropy estimation". In: *arXiv preprint arXiv:0908.3961* (2009).

[47] Ashwin Lall et al. "Data streaming algorithms for estimating entropy of network traffic". In: *ACM SIGMETRICS Performance Evaluation Review* 34.1 (2006), pp. 145–156.

[48] Pankaj K Agarwal et al. "Mergeable summaries". In: *ACM TODS* (2013).

[49] Behnaz Arzani et al. "PrivateEye: Scalable and Privacy-Preserving Compromise Detection in the Cloud". In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 797–815.

[50] Changhoon Kim et al. "In-band Network Telemetry via Programmable Dataplanes". In: *Deme session of ACM SIGCOMM*. 2015.

[51] Ulrike von Luxburg. *A Tutorial on Spectral Clustering*. 2007. arXiv: 0711.0189 [cs.DS].

[52] Ran Ben Basat et al. "Constant time updates in hierarchical heavy hitters". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 2017, pp. 127–140.

[53] *ConnectX® Ethernet Adapters*. URL: https://www.mellanox.com/products/ethernet/connectx-smartnic (visited on 04/28/2020).

[54] *Introduction to Cache Allocation Technology in the Intel® Xeon®...* URL: https://software.intel.com/content/www/us/en/develop/articles/introduction-to-cache-allocation-technology.html (visited on 09/17/2020).

[55] *Constraints*. URL: https://www.gurobi.com/documentation/9.1/refman/constraints.html#subsubsection:GenConstrSimple (visited on 03/05/2021).

[56] *Non-Convex Quadratic Optimization - Gurobi*. URL: https://www.gurobi.com/resource/non-convex-quadratic-optimization (visited on 03/05/2021).

[57] *Benchmarking Performance and Scaling of Python Clustering Algorithms — hdbscan 0.8.1 documentation*. URL: https://hdbscan.readthedocs.io/en/latest/performance_and_scalability.html (visited on 09/16/2020).

[58] Dimitris Bertsimas, John Tsitsiklis, et al. "Simulated Annealing". In: *Statistical science* 8.1 (1993), pp. 10–15.

[59] *The Pktgen Application — Pktgen 3.2.4 documentation*. URL: https://pktgen-dpdk.readthedocs.io/en/latest (visited on 05/01/2020).

[60] *Testpmd Application User Guide — Data Plane Development Kit 20.05.0 documentation*. URL: https://doc.dpdk.org/guides/testpmd_app_ug (visited on 05/26/2020).

[61] S. Knight et al. "The Internet Topology Zoo". In: *Selected Areas in Communications, IEEE Journal on* 29.9 (Oct. 2011), pp. 1765–1775.

[62] Ankit Singla et al. "Jellyfish: Networking data centers randomly". In: *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. 2012, pp. 225–238.

[63] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. "A Scalable, Commodity Data Center Network Architecture". In: *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*. SIGCOMM '08. Seattle, WA, USA: Association for Computing Machinery, 2008, pp. 63–74.

[64] frenetic-lang. *ocaml-topology*. URL: https://github.com/frenetic-lang/ocaml-topology (visited on 05/26/2020).

[65] Lun Li et al. "A First-Principles Approach To Understanding the Internet's Router-Level Topology". In: *ACM SIGCOMM Computer Communication Review* 34.4 (2004), pp. 3–14.

[66] *Structure & Strangeness*. URL: https://www.cs.unm.edu/~aaron/research/fastmodularity.htm (visited on 03/05/2021).

[67] Xiaoqi Chen et al. "BeauCoup: Answering Many Network Traffic Queries, One Memory Update at a Time". In: *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*. SIGCOMM '20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 226–239.

[68] Masoud Moshref et al. "SCREAM: Sketch Resource Allocation for Software-Defined Measurement". In: *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*. CoNEXT '15. Heidelberg, Germany: Association for Computing Machinery, 2015.

[69] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. "Distributed network monitoring and debugging with switchpointer". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 2018, pp. 453–456.

[70] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. "Confluo: Distributed monitoring and diagnosis stack for high-speed networks". In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 2019, pp. 421–436.

[71] Thomas Holterbach et al. "Blink: Fast connectivity recovery entirely in the data plane". In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 2019, pp. 161–176.

[72] Chang Lou, Peng Huang, and Scott Smith. "Understanding, Detecting and Localizing Partial Failures in Large System Software". In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 2020, pp. 559–574.

[73] Wei Le and Mary Lou Soffa. "Marple: a demand-driven path-sensitive buffer overflow detector". In: *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. 2008, pp. 272–282.

[74] Mojgan Ghasemi, Theophilus Benson, and Jennifer Rexford. "Dapper: Data plane performance diagnosis of tcp". In: *Proceedings of the Symposium on SDN Research*. 2017, pp. 61–74.

[75] Firas Abuzaid et al. "Contracting Wide-area Network Topologies to Solve Flow Problems Quickly". In: *NSDI*. USENIX. Nov. 2020.

[76] Mingran Yang et al. "Joltik: Enabling Energy-Efficient "Future-Proof" Analytics on Low-Power Wide-Area Networks". In: *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*. MobiCom '20. London, United Kingdom: Association for Computing Machinery, 2020.

[77] *Working With Multiple Objective*. URL: https://www.gurobi.com/documentation/9.0/refman/working_with_multiple_obje.html (visited on 09/17/2020).

(a) Spectral Clustering on Topology **(b) Spectral Clustering on Communication Graph** **(c) Clustering using multi-tenant heuristic**
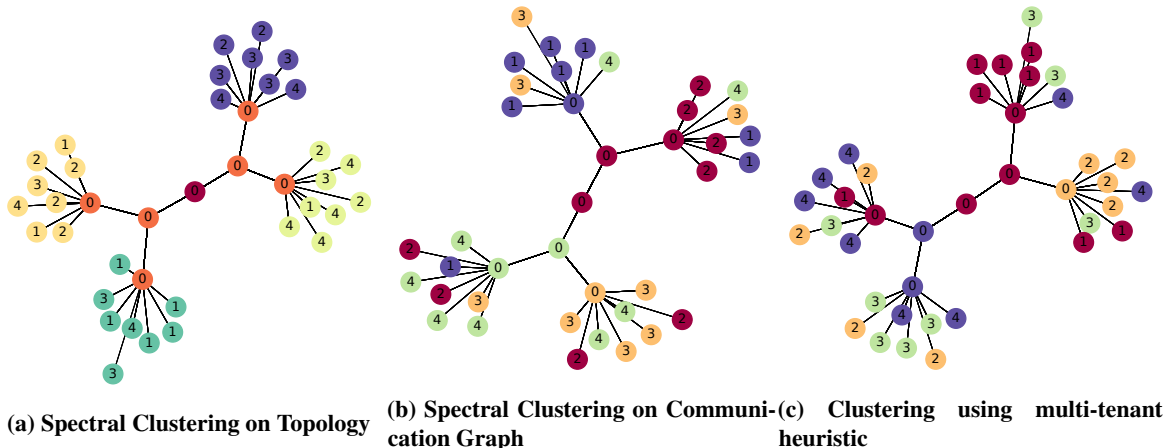
Figure 11: Output of clustering

A Clustering Details

Clustering Heuristic. We performed simulated annealing to explore the space of clustering solutions. Figure 12 shows the annealing in action. We find that there are many clustering solutions that result in optimization solutions which are close to optimal.
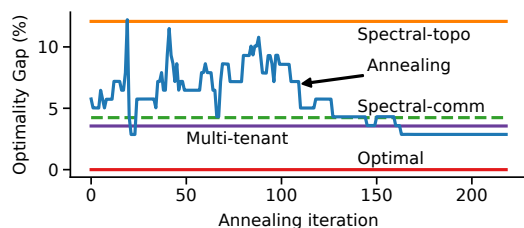
Figure 12: Simulated Annealing — Spectral-topo refers to spectral clustering over network topology. Similarly, Spectral-comm is over the communication graph. Multi-tenant refers to our domain specific heuristic. Optimal refers to no clustering.

Clustering Output. Figure 11 illustrates examples of the clustering output for different clustering techniques. The topology shown is a simple tree topology with only servers and switches (without NICs) for ease of visualization. The nodes marked '0' are switches and the other nodes are servers. The non-zero numbers on the servers signify that servers with the same number are communicating with each other. The colors signify that devices having the same color are in the same cluster.

Optimality vs. Solving Time Trade-off. Figure 7b and 7a also suggest that our clustering technique does not significantly degrade resource efficiency. This is counter-intuitive since clusters limit the types of optimization possible. We explore this in in Figure 13. Cluster size represents a trade-off between optimality and solving time, i.e. smaller clusters help reduce solving time at the cost of optimality. We observe

that the even for relatively small clusters (20 devices), the optimality gap is very low ($< 0.4\%$), allowing us to choose small clusters to reduce the run-time while preserving close to optimal solutions. The extreme case of all devices within the same cluster mimics the case of no clustering. In this case, all sketches would be assigned to the only available cluster and then the Optimizer is run to place sketches on the devices within that single cluster, effectively deciding between all devices of the topology.

The MIP solver also natively allows trading-off solving time for optimality through configuration of a time limit. In Figure 7a, we configured a time limit of 300s. We find that HeteroSketch is able to produce better quality solutions in lesser time, achieving a better trade-off between optimality and solving time.
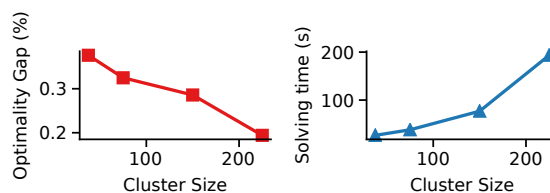
Figure 13: Cluster size — optimality gap vs runtime. The optimality gap is the gap between the objective of HeteroSketch relative to the objective value of Optimal. (For experiment with $pods = 24$).

Choosing Cluster Size. As we see in Figure 13, the clustering heuristic provides a trade-off between solution quality (optimality) and optimization run-time. We want to be able to select the largest cluster size which allows an acceptable run-time. To do this, we look at the knee of the graph between solving time and number of devices (Figure 14). Since, the solving time depends not only on the network topology but also on the monitoring load (Y, see §7.2), we need to recompute this graph whenever the monitoring load changes significantly. We use the following procedure to quickly re-

compute the solving time vs number of devices graph: we divide the network topology into clusters of different sizes. Then we run the Optimizer for a sample of these clusters which have different sizes, and we effectively obtain solving time as a function of number of nodes (cluster size). Figure 14 shows this in action.
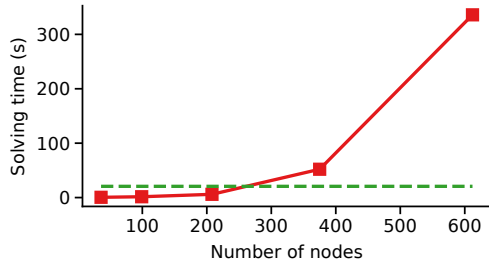
Figure 14: Determining cluster sizes — X-axis represents the number of nodes in a cluster, and Y-axis denotes maximum time to solve *just one* cluster (averaged over 5 clusters of the same size in the topology)

B Performance Profiler Details

Non-uniform Memory Access Pattern. Sketches such as UnivMon which have multiple control paths can result in non-uniform access of the working memory set even if the traffic is uniform. There are at least the following two ways to handle such cases: (1) estimating *effective memory size* that is accessed uniformly, (2) estimating *hit rates* to different levels of memory hierarchy. In what follows provide some background on the operations of the UnivMon sketch and then describe these two approaches in more detail. In our implementation, we use the first approach to incorporate UnivMon.

- *Background on UnivMon sketch.* UnivMon is an ensemble of Count Sketches and consists of multiple levels. Each level maintains a Count Sketch. On every packet, a hash function is computed to decide a level and the corresponding level is updated.[3] The level is decided by the count of leading non-zero bits of the hash output. Each bit of the hash output is equally likely to be zero or one. Due to this, subsequent levels are accessed with exponentially decreasing probability, i.e.,

$$P(i) = \begin{cases} 2^{-i} & \text{if } i < k \\ 2^{-(k-1)} & \text{if } i = k \end{cases} \quad (2)$$

where P is the probability of accessing level i, and k is the total number of levels.

- *Effective memory size accessed uniformly.* As shown in Figure 15, we study the variation of time per packet

[3]Note, we use an optimized version of UnivMon described in [76] which is slightly different from the original UnivMon paper [11]
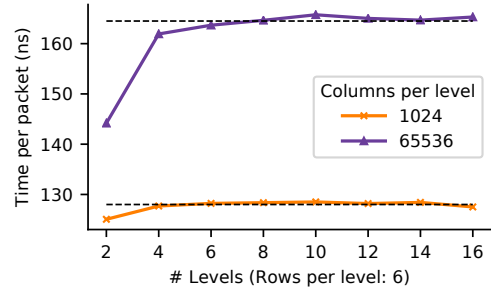
Figure 15: Dealing with non-uniform memory access patterns UnivMon effectively behaves as having at most 4 levels from the perspective of memory size accessed uniformly. The black dotted lines show the time per packet for a hypothetical sketch with $2 * 6 + 1 = 13$ hashes per packet and 6 memory accesses where the accesses are made uniformly. (The 13 hashes are: 2 hashes per row for a Count Sketch and one hash to decide level.)

for the UnivMon sketch on a CPU as we vary its levels. We observe that UnivMon effectively behaves as if it had at most four levels, all of which are accessed uniformly. In the Optimizer, this corresponds to using T(Effective uniformly accessed memory) instead of T(Total memory) (T was defined in §4). For other sketches with complex control paths, a similar strategy can be used.

- *Estimation of hit rates.* This is a more theoretical approach, but has similar results as above method. We assume that the caching mechanism on the device being profiled honors temporal locality to decide which items to keep cached, i.e., we can assume that the probability that an item is kept in the cache is proportional to its access frequency. Using this, we estimate the likelihood that an item is cached. Then using (1) the likelihood that an item is accessed (access frequency) and (2) the likelihood that the item is in cache; we estimate the expected inverse throughput of memory accesses leveraging inverse throughputs to different cache levels. Inverse throughputs to cache levels are estimated using the ridges in the memory benchmark of the Profiler (Figure 4a). In what follows, we illustrate application of this process on UnivMon with a toy device.

Let's assume the UnivMon sketch has k levels and each level has consumes x bytes. For simplicity, let's assume the toy device has two cache levels: L1 and L2, with sizes x and $(k-1)x$ bytes respectively. Recall that, the ith level of UnivMon is accessed with probability $P(i)$. Through the locality principle, we expect that $P(i)$ fraction of level i would be present in L1 cache and $1 - P(i)$ in L2 cache. Then the probability that an access goes to

L1 cache is:

$$P_{L1} = \sum_{i=1}^{k} P_1 * P_2 \qquad (3)$$

$$= P(i) * P(i)$$

$$= \left(\sum_{i=1}^{k-1} 2^{-2i} \right) + 2^{-2(k-1)}$$

$$= \frac{1}{4} \left(1 + \left(\sum_{i=2}^{k-1} 2^{-2(i-1)} \right) + 2^{-2(k-2)} \right)$$

$$\approx \frac{1}{4}$$

$$\text{and, } P_{L2} = 1 - P_{L1} \approx \frac{3}{4}$$

where P_1 is the probability that level i of the sketch is accessed, P_2 is the probability that the accessed data is in L1 cache given that some data in level i is accessed, and P_{L2} is the probability that L2 is accessed. Then, the expected inverse throughput of memory accesses would be $P_{L1} * IT(L1) + P_{L2} * IT(L2)$, where $IT(.)$ is the inverse throughput to the corresponding cache level.

For the same toy device, we do a similar analysis for a sketch with size $4x$ bytes, where the memory accesses are made uniformly. Then random x bytes of the sketch would be in L1 and remaining $3x$ in L2. $P_{L1} = $ probability that an element in L1 cache is accessed $= x/4x$. Hence, expected inverse throughput for memory accesses is $0.25 * IT(L1) + 0.75 * IT(L2)$. We see that the access probabilities (hit rates) and the expected inverse memory access throughput for this sketch is roughly equal for Univ-Mon example above, consistent with the empirical approach.

Such expressions for expected memory access time can be accommodated as constraints in the MIP formulation of the Optimizer albeit with increased solving time due to additional non-linear constraints. We leave exploration of this for future work.

Non-uniform Traffic. In the Optimizer and Profiler, we study performance and allocate resources for worst case (uniform) traffic. This is because the traffic distributions may not be known apriori, or one might want to allocate resources to handle adversarial cases. However, if this is not true, one could adapt our work to use the known traffic distribution to estimate hit rates to different cache levels similar to that described for accommodating sketches with non-uniform memory access patterns above.

Impact of profiling errors on Optimizer. We observe that most of the errors in profiling occur when sketches use a large amount of DRAM (or off-chip memory) on the devices. We seek to study what difference, such errors can create to the Optimizer's output. In the Optimizer's output, we identify devices which use a non-zero amount of DRAM, and whose resource allocation would change if the device profiles are off

by 10%. We show in Figure 16 how many more resources would be needed for a Clos topology with 16 pods as we vary the sketch load (Y) (defined in §7.2). We observe that consistently less than 5% of devices satisfy the above conditions. Hence, the Optimizer's allocation would be off only for these 5% of devices. To illustrate this, let's assume all 5% devices are CPUs, each of these devices would need one more core if the profiles under-predict time per packet. Our savings suggest 0.5(50%) to 1(100%) cores saved per server. The errors are significantly smaller compared to the demonstrated savings. Note, that the profiles still are assumed to be accurate for the cases when sketches don't occupy DRAM.
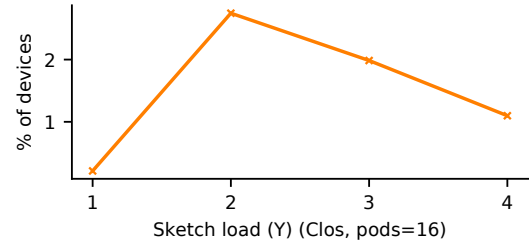
Figure 16: Impact of profiling errors The Y-axis shows the percentage of devices which use DRAM and whose resource allocation would change if the profiles are off by 10%.

Supplementary Evaluation. In addition to the Profiler evaluation for Count-Min sketches on CPUs, shown in Figure 6, we have also have the the detailed results for the other sketches including Count Sketch and UnivMon on CPU, SmartNIC, and FPGA shown in Figures 17, 18, and 19.

C Optimizer Details

Device profiles. Here we give an example of what the device profile ($dtime(.)$) looks like. The following shows the device profile for SoC SmartNIC.

$$dtime = max \left(\frac{k_1 + u_h \cdot h}{c}, k_2 + u_m \cdot T(m) \right) \qquad (4)$$

$$= max(dtime_h, k_2 + u_m \cdot T(m))$$

$$= max(dtime_h, k_2 + u_m \cdot t_{mem})$$

$$dtime_h \cdot c = k_1 + u_h \cdot h \qquad (5)$$

$$t_{mem} = T(m) \qquad (6)$$

where u_h, u_m, and m (as defined in Section 4) are expressed as linear functions of sketch placement decision variables and c is a decision variable corresponding to the number of micro-engines. $T(.)$ is a non-linear function modelled using piecewise-linear constraints and the product terms: $u_m \cdot t_{mem}$, $dtime_h \cdot c$ are modelled using bi-linear constraints. These functions show a decoupled system with sketching done on the forwarding critical path with fraction of parallelizable execution $f \approx 1$.

Tweaking MI-BLP. We show how the MI-BLP formulation can be adapted to support other objectives/goals. We show in Equation 7, how a user can minimize performance overhead as an objective and subject to this minimum, again minimize resource overhead. This is done using hierarchical objectives [77].

$$
\begin{aligned}
\text{O1': perf} \quad & \text{Minimize} \max_{d \in \mathcal{D}} (time_d), & (7) \\
\text{O2': resources} \quad & \text{Minimize} \sum_{d \in \mathcal{D}} (res_d + mem_d), \quad \textbf{s.t.} \\
\text{C1: coverage} \quad & \sum_{d \in p_\pi} b_{(d,s)} \geq 1 \quad \forall p \in \mathcal{P}, \forall s \in p_s \\
\text{C2: accuracy} \quad & mem_{(d,s)} \geq s_{mem} \cdot b_{(d,s)} \quad \forall s \in \mathcal{S}, \forall d \in \mathcal{D} \\
\text{C3: capacity} \quad & \sum_{s \in \mathcal{S}} b_{(d,s)} \cdot s_{rows} \leq d_{rows}, \quad \text{and} \\
& mem_d = \sum_{s \in \mathcal{S}} mem_{(d,s)} \leq d_{mem} \quad \forall d \in \mathcal{D} \\
\text{C4: profiles} \quad & \forall d \in \mathcal{D}: \\
& time_d = d_{time}(res_d, \mathcal{P}_d, \\
& \qquad \{(mem_{(d,s)}, b_{(d,s)}) | s \in \mathcal{S}\} \\
\text{C5: traffic} \quad & time_d \leq \frac{1}{d_{\text{traffic}}} \quad \forall d \in \mathcal{D}, \quad \text{where} \\
& d_{\text{traffic}} = \sum_{p \in \mathcal{P}_d} p_t, \quad \mathcal{P}_d = \{p | d \in p_\pi, p \in \mathcal{P}\}
\end{aligned}
$$

Supplementary Evaluation.

- We show additional metrics collected for Figures 7a and 7b in Figures 20 & 21.

- One of the ways in which HeteroSketch reduces resource overhead is through efficient use of resources that it allocates. We see in Figures 8a, 8a, and 8a, that HeteroSketch overall allocates less number of resources but better utilizes each resource that it does allocate including CPU cores, SoC NIC memory bandwidth, micro-engines on the SoC smart-NIC.
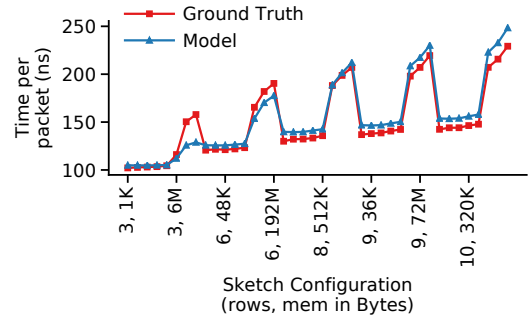
(a) 2 Cores, Count Sketch
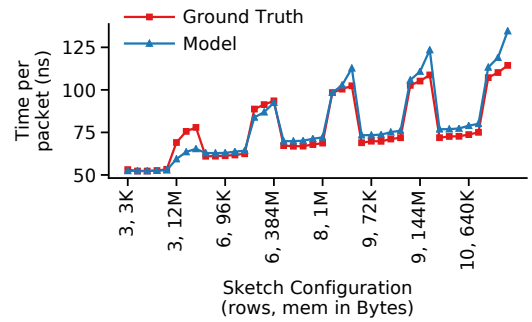
(b) 4 Cores, Count Sketch
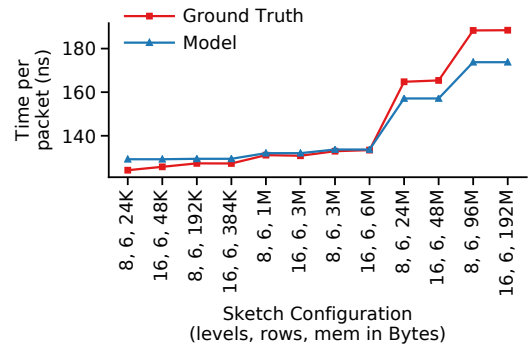
(c) 2 Cores, UnivMon
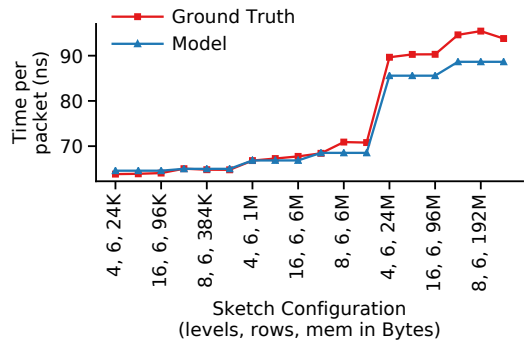
(d) 4 Cores, UnivMon

Figure 17: Performance Profiler — CPU Model
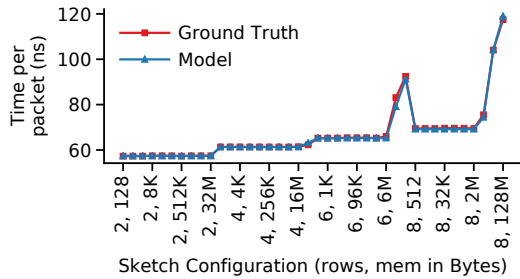
(a) 36 Micro-engines, Count-Min Sketch
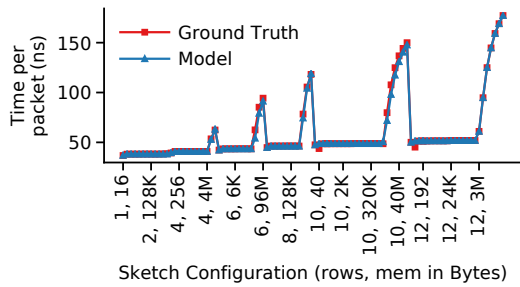
(b) 54 Micro-engines, Count-Min Sketch
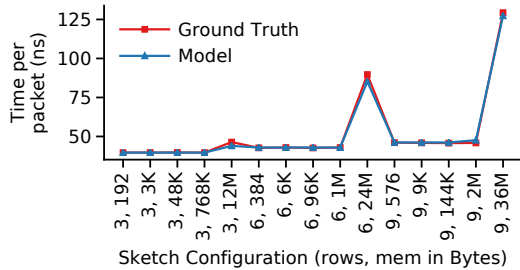
(c) 54 Micro-engines, Count Sketch
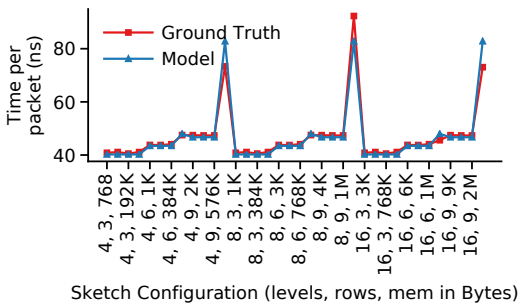
(d) 54 Micro-engines, UnivMon

Figure 18: Performance Profiler — SmartNIC Model
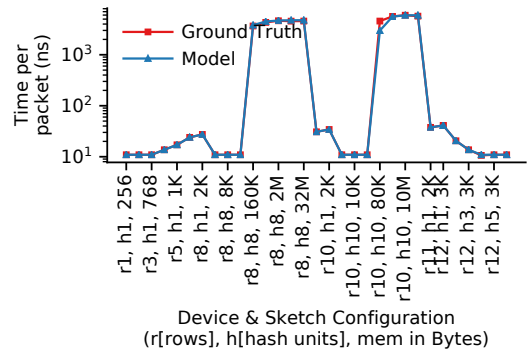
(a) Count-Min Sketch
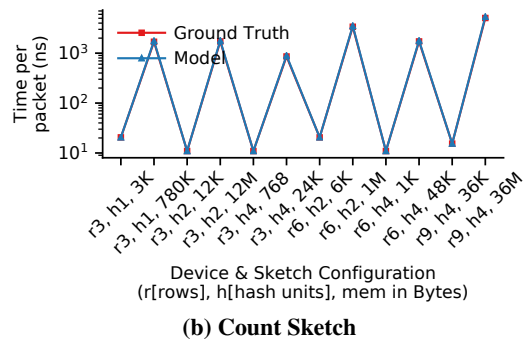
(b) Count Sketch
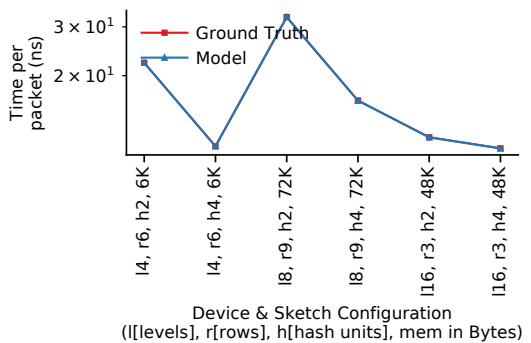
(c) UnivMon

Figure 19: Performance Profiler — FPGA Model (Note: Y-axes are in log-scale)
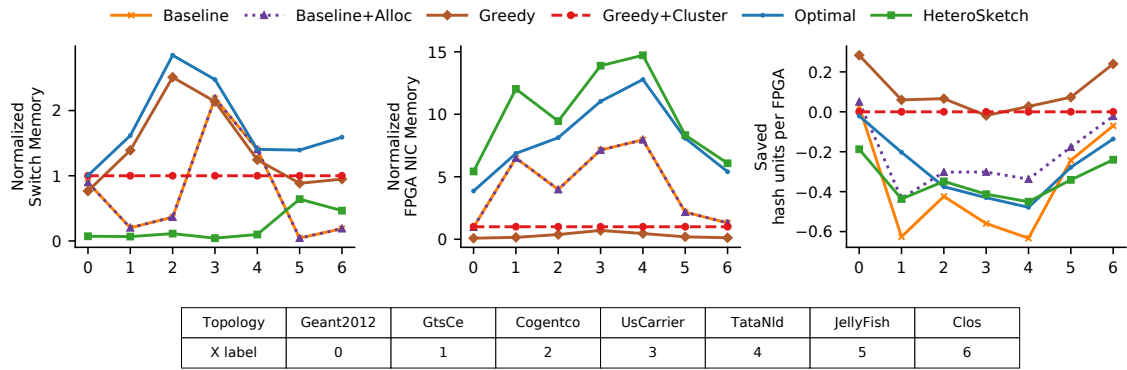
Figure 20: Supplementary Optimizer Evaluation — [Topology Study] These figures show the difference in resource usage for experiments in Figure 7a in terms of switch & FPGA memory, and FPGA hash unit instances. Compute resources are shown in terms of amount saved relative to Greedy+Cluster (negative compute resources implies more resource consumption than Greedy+Cluster). Total resources and memory resources are normalized w.r.t Greedy+Cluster.
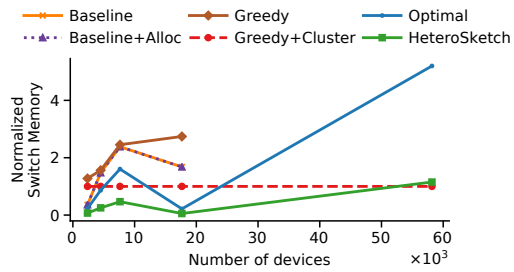
Figure 21: Supplementary Optimizer Evaluation — [Scale Study] These figures show the difference in resource usage for experiments in Figure 7b in switch memory normalized by that of Greedy+Cluster.

SketchLib: Enabling Efficient Sketch-based Monitoring on Programmable Switches

Hun Namkung[⋆], Zaoxing Liu[†], Daehyeok Kim[⋆§], Vyas Sekar[⋆], Peter Steenkiste[⋆]

[⋆]*Carnegie Mellon University,* [†]*Boston University,* [§]*Microsoft*

Abstract

Sketching algorithms or sketches enable accurate network measurement results with low resource footprints. While emerging programmable switches are an attractive target to get these benefits, current implementations of sketches are either inefficient and/or infeasible on hardware. Our contributions in the paper are: (1) systematically analyzing the resource bottlenecks of existing sketch implementations in hardware; (2) identifying practical and correct-by-construction optimization techniques to tackle the identified bottlenecks; and (3) designing an easy-to-use library called *SketchLib* to help developers efficiently implement their sketch algorithms in switch hardware to benefit from these resource optimizations. Our evaluation on state-of-the-art sketches demonstrates that SketchLib reduces the hardware resource footprint up to 96% without impacting fidelity.

1 Introduction

The ability to monitor network traffic is necessary for various network management tasks such as traffic engineering, anomaly detection, load balancing, and resource provisioning [10, 13, 27, 29, 43, 45, 54]. In this respect, recent developments in programmable switches and attendant languages [9, 14] make it possible to support richer fine-grained and real-time monitoring capabilities.

With this network programmability, sketch-based monitoring has emerged as a promising alternative to traditional sampling-based techniques [19, 49]. At a high-level, sketch algorithms consist of updating multiple counter arrays with a series of independent *hash function calls* and *counter updates*. Sketch-based approaches have been developed to support a broad spectrum of measurement tasks with provable resource-accuracy trade-offs, including heavy-hitter detection or quantile estimation (e.g., [17, 21]), general estimation capabilities (e.g., UnivMon [41]), and more expressive multidimensional analytics (e.g., R-HHH [12]).

While prior efforts have demonstrated the feasibility of expressing sketches using these language APIs [32, 41, 46, 53], implementing sketches efficiently in hardware remains an open challenge. For example, off-the-shelf sketch implementations often cannot run with the desired accuracy levels due to insufficient hardware resources (see §3). Indeed, some proposed sketches (e.g., [41]) are infeasible as implemented, or even if they are feasible, consume significant resources.

Even if more hardware resources may become available, so too do operators' demands of in-switch applications, and the resources consumed by sketches will be unavailable for other switch functions. Thus, it is essential to explore if, and how, we can efficiently realize sketch-based telemetry on programmable switches. This is the central question that this paper tackles. Specifically, we focus on programmable hardware switches based on the Reconfigurable Match-Action Tables (RMT) paradigm [1].

We identify and analyze four key resource bottlenecks for realizing sketches on RMT switch hardware:

- *Hash calls:* Sketches make a number of counter updates based on independent hash functions, requiring a large number of hash calls in hardware.
- *Memory accesses:* Sketches need to access on-chip memory (e.g., SRAM) for counter updates, but the number of memory accesses per packet is limited in hardware.
- *Pipeline stages:* Some sketches need to select a subset of counter arrays for counter updates [23, 37, 41]. However, implementing this naively can cause a long chain of sequential computation dependencies which stresses the limited number of switch pipeline stages.
- *Resources for tracking heavy flowkeys:* Some sketches need to keep track of the flowkeys identifying the heavy hitters (e.g., 5-tuple, source IP, or destination IP) [12, 17, 21, 36, 41]. Common structures such as priority queues or heaps used in software are not supported on programmable switches and existing solutions entail undesirable tradeoffs between miss rate, data plane memory, and control plane bandwidth.

Having identified these bottlenecks, our contribution is a careful synthesis of known and novel optimizations into a practical library for enabling efficient sketch implemen-

tations atop the RMT architecture. While some of these build on prior work in optimizing sketching for other targets such as software switches, FPGAs, and embedded platforms [40, 51, 52, 55], our main contribution is in realizing feasible and effective optimizations based on our bottleneck analysis and translating them into the switch hardware setting. For example, to reduce the number of hash calls, we identify opportunities to consolidate and reuse hash results across multiple counter updates [24, 35]. Similarly, we identify an opportunity to reduce the pipeline stages by eliminating code dependencies based on longest prefix matching using TCAM [55]. We reduce the memory accesses by refactoring sketch algorithms and removing unnecessary memory accesses. We also develop practical flowkey tracking mechanisms that are feasible in hardware. Note that all optimizations preserve correctness while reducing the resource footprint.

To make it easy for sketch developers to benefit from these optimizations with minimal effort, we implement *Sketch-Lib*, an easy-to-use API using the P4 language [14]. These optimizations can be applied to a broad spectrum of classical sketches (e.g., [17, 21, 36]) and recent innovations (e.g., [12, 41]). We qualitatively evaluate the suitability of SketchLib for 19 published sketches and observe that 15 of them can be expressed and can benefit from one or more of our optimizations. We acknowledge that not all optimizations are applicable for every sketch and we envision sketch developers using our API to adopt the relevant optimizations.

We quantitatively evaluate the utility of SketchLib in improving 7 of the 15 applicable sketches covering a diverse set of target telemetry tasks: Count Sketch (CS) [17], PCSA [25], MRAC [37], Multi-resolution Bitmap [23], Hierarchical Heavy Hitters [12], and UnivMon [41]. Our evaluation using a range of packet traces empirically confirms that our optimizations provide similar accuracy ($\leq 1.9\%$) with substantially (up to 96%) reduced resource usage. Furthermore, some complex sketches (e.g., UnivMon) that were previously infeasible on current hardware become feasible.

Contributions and Roadmap. To summarize, we make the following contributions:

- **Bottleneck Analysis (§3):** We identified four key resource bottlenecks for sketch implementations on the hardware programmable switch.
- **Optimizations (§4):** We identify and synthesize practical correctness-preserving optimizations to address the bottlenecks for sketches on switch hardware.
- **API Implementation (§5):** We design a convenient API to make our optimizations easy to use for developers who implement sketches on RMT programmable switches.[1] We verified significant resource benefits on a broad range of sketching algorithms.

---

[1]SketchLib is publicly available at https://github.com/SketchLib.
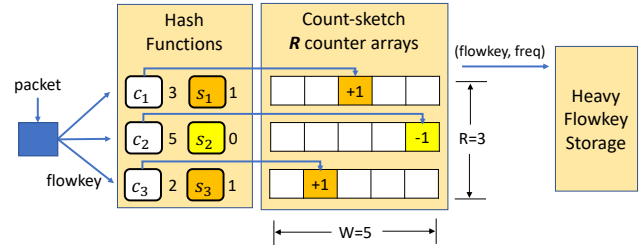
Figure 1: Count Sketch has three components - hash computations, multiple counter arrays, and heavy flowkey storage.

(a) R-HHH   (b) UnivMon

Figure 2: Simplified P4 code of existing multi-level sketches.

2 Background

In this section, we start by providing some background on sketching algorithms and programmable switch architecture. We then describe how the sketch code is mapped onto the hardware resources.

2.1 Background on Sketches

Sketching algorithms or *sketches* are randomized approximation algorithms that are designed to compute different observed statistics on a given data stream during every measurement time interval called *epoch*. In network monitoring, prior work has shown that sketches (e.g., [12,17,21,32,40–42, 46,53]) offer better resource-accuracy tradeoffs relative to traditional techniques that rely on sampling (e.g., NetFlow [19]). Our focus in this paper is not to develop new sketches but to enable efficient sketch realizations on programmable switches. To better understand the different resource requirements of sketches, we classify prior sketching work into two categories:

1. Single-level sketches: As a canonical example, consider the *count sketch* (CS) [17] for heavy hitter detection shown in Fig. 1. A single-level sketch such as Count Sketch maintains a 2D-array of counters: R independent counter arrays with size of W; i.e., $R \times W$ memory counters. As each packet arrives, we extract a flowkey from the packet (e.g., srcIP, IP 5-tuple). On this key, we compute two independent hash functions c_i and s_i, corresponding for each row i. c_i is used to select a specific column and s_i is a 1-bit hash used to determine either to increase or decrease the counter.
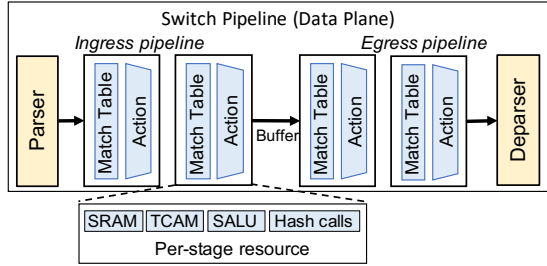
Figure 3: RMT switch architecture.

Figure 4: Mapping P4 code to switch resources.

The total number of hash computations is $2R$. Count Sketch requires additional memory space for storing heavy flowkeys whose estimated flow counts are above a threshold. Other single-level sketches requiring a 2D-array include the count-min sketch (CMS) [21], k-ary (Kary) sketch [36]. Some single-level sketches like HyperLogLog (HLL) [26] only need a 1D array data structure.

2. Multi-level sketches: Conceptually, these consist of multiple single-level sketches to enable richer queries. For instance, R-HHH and UnivMon can use multiple count sketches, called levels (e.g., L levels of $R \times W$ counters). R-HHH supports detection of hierarchical heavy hitters, which detects heavy hitters based on different lengths of IP prefixes and Univ-Mon provides more general estimation capabilities. Other sketches like PCSA, MRAC, and multi resolution bitmap (MRB) [23,25,37] use multiple 1D-array single-level sketches. Multi-level sketches typically select a subset of counter arrays to issue counter updates for a given flowkey. For instance, as shown in Fig. 2a, R-HHH randomly selects one level of count sketch using a level-specific key (e.g., IP prefix) to update per packet. In contrast, UnivMon uses an additional sampling stage using hash functions that return 0 or 1 to select levels for update, as shown in Fig. 2b.

2.2 Programmable Switch Hardware

Our focus in this paper is programmable switch hardware based on the Reconfigurable Match-Action Tables (RMT) paradigm [15]. A canonical commercial realization of this architecture is the Intel Tofino switch chip [1]. Based on public documentation and conversations with vendors, we believe that while other programmable switches (e.g., Broadcom Trident [2]) may have different hardware resource allocation strategies, the architectural bottlenecks for sketches are likely similar. We leave it as future work to extend SketchLib to other hardware targets.

Hardware architecture. RMT-based programmable switches have a pipeline of reconfigurable match-action tables in the data plane, as shown in Fig. 3. There are constraints in packet processing pipeline to meet the line-rate processing requirement. For example, at each stage, a packet can access a limited amount of compute and memory resources. Each stage has an identical design with the same types of resources. To provide flexible match-action operations, each stage has a *match table* that matches packet
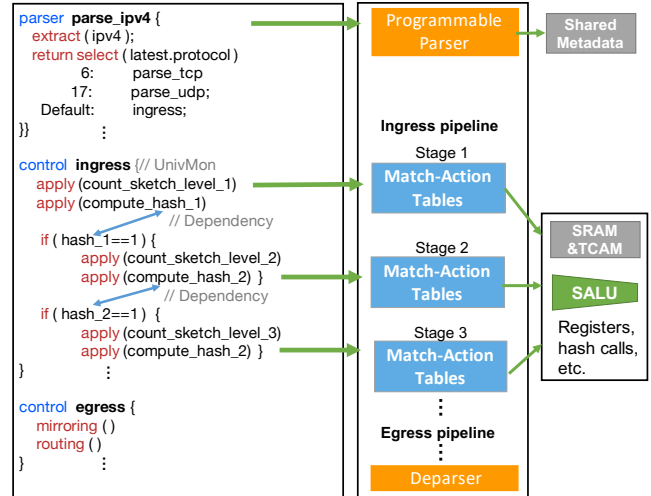
headers to specific values followed by an action unit that executes a set of simple instructions, depending on the output of the matching unit.

Key hardware resources. We now briefly describe the key hardware resources available in each pipeline stage. First, there are a number of hardware *hash function calls* (hash calls) per pipeline stage. They are used to compute hash functions (e.g., CRC with user-defined polynomials) over packet header fields or metadata to support operations such as load balancing and table lookups. Each pipeline stage also has a fixed amount of SRAM that can be used to maintain state, for example counter arrays. *Stateful ALUs* (SALUs) are hardware resources that allow one read and one write operation to the stateful object in SRAM. Each SALU can be used for counter update operations such as counter increment or decrement. Finally, each pipeline stage is also equipped with some amount of *ternary content-addressable memory* (TCAM) that can be used for wildcard matches over header fields. Overall, the amount of these resources is fixed at hardware design time, and it is limited. For example, a commercial programmable switch today is equipped with (at most) 10 SALUs, 10 hash calls, 10 MBs of SRAM and TCAM per pipeline stage with a total of 12 pipeline stages [15,43,56].[2]

The data plane can interact with the switch control plane for additional processing. However, the switch control plane is not designed for real-time processing, e.g., the bandwidth to the control plane is limited and the response time is high. So it is only useful for infrequent operations.

2.3 P4 Programming and Compilation

Programs for RMT switches are written in the P4 language [14] as illustrated in Fig. 4. At a high-level, a P4 program consists of the following components. First, a packet parser parses the header fields of each packet and stores the extracted fields into metadata. Second, a series of match-action

---

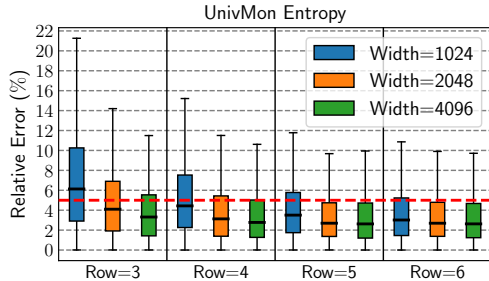[2]The other absolute resource numbers are proprietary.

Figure 5: UnivMon entropy estimation error for different configurations. Dotted red line indicates target accuracy.

operations are executed based on the *match-action* abstraction, e.g., matching a specific header field and update a register as an action. The action is specified by special functions that map operations to hardware resources, e.g., functions for hashing and accessing memory. Finally, the P4 program defines the packet forwarding behaviors, e.g., routing a packet to an egress port, recirculating it in the pipeline, or forwarding it to the switch control plane.

The P4 compiler maps the P4 program into a static pipeline realization. The compiler analyzes the *dependencies* between operations in the P4 program, to map the program on to the pipeline stages. For instance, given the code snippet in Fig. 4, the resolution of each `if`-clause depends on the previous hash result. Because of this dependency, two consecutive `if`-clauses cannot run in parallel, so the compiler has to map them to different pipeline stages in order for them to run sequentially. If a mapping of a whole program is possible considering hardware constraints, packets are guaranteed to be processed at line rate; otherwise compilation fails. Note that vendor-specific compiler backends are typically proprietary.

3   Bottleneck Analysis

In this section, we consider three exemplar sketches (single-level: count sketch; multi-level: R-HHH and UnivMon) to quantify the resource bottlenecks. We implement them in P4 based on the logic described in prior work [17, 23, 25, 26, 37, 41] similar to the structure presented in Fig. 2.

3.1   Methodology and Setup

Configuring sketches. Running sketches entails picking parameters (e.g., the count (R) and size (W) of counter arrays) to trade-off the accuracy vs. resource use. We envision an operator configuring the sketches with some target accuracy goal, e.g., the median error should be less than 5%. Operators can use trace-driven analysis to pick reasonable operating regimes for these parameters.

As an example, Fig. 5 illustrates this trade-off for entropy estimation using UnivMon. The figure shows the estimation accuracy using an hour-long inter-ISP packet trace captured on a OC-192 link [7] with different parameters R and W for count sketches and $L = 16$ levels. We see that the error decreases as we increase the number of rows (R) and width (W)

for count sketches. Naturally, the higher accuracy configurations incur more hardware resources. For our bottleneck analysis, we target an accuracy of under 5% median error (dotted red line in Fig. 5), which we achieve with minimal resource use with the configuration $R = 3$ and $W = 2048$. We repeat the analysis for count sketch and R-HHH and consider a similar operating regime for these sketches as well.

Estimating resource footprint. For a given set of sketch parameters, the most direct way to measure the required hardware resources is to compile the code and run it on the hardware. However, this limits our analysis to currently available platforms. In order to support "what if" analysis for hardware with different resources (e.g., more pipeline stages), we extended an existing open source tool for mapping P4 programs to the RMT hardware, which we will refer to as RMT resource mapper [34]. Specifically, we address three issues to extend RMT resource mapper for our analysis:

- *Inputs:* The input to Tofino compiler is P4-16 code with some hardware-specific primitives whereas RMT resource mapper accepts only P4-14 code [8]. Thus, we first convert our P4-16 code into equivalent P4-14 code. Then, we convert Tofino-specific primitives to equivalent ones specified in the language specification. For instance, we replace Tofino-specific primitives for accessing registers with *register_read* and *register_write*.

- *Resources:* First, RMT resource mapper does not model hash calls and SALUs in their original design. Thus, we extend RMT resource mapper to model hash calls and SALUs and added the corresponding optimization constraints for assignment of these new resources. Second, we observed that RMT resource mapper assigns memory even for tables without any entries and action data. To fix this disconnect, we decouple the memory/table assignment.

- *Objective:* RMT resource mapper supports different optimization objectives: minimizing latency, power, or pipeline stages. The objective of minimizing pipeline stages is the most suitable because it gives resource mappings that are closest to those generated by the Tofino compiler.

With these fixes in place, we validate our extensions by comparing the resource usages between RMT resource mapper and Tofino compiler for a wide range of sketches and configurations, for the cases that are feasible on current hardware. Based on the measurement results, we conclude that our modified RMT resource mapper is a good proxy of Tofino compiler as it captures the relevant resource constraints, and its resource allocation results are close to that of Tofino compiler (see Appendix A for more details).

3.2   Identified Bottlenecks

Using the RMT resource mapper, we measure the usage of each type of resources based on the output of the compiler for three sketches: Count Sketch, R-HHH, and UnivMon. For the purpose of bottleneck analysis, we use a base configuration
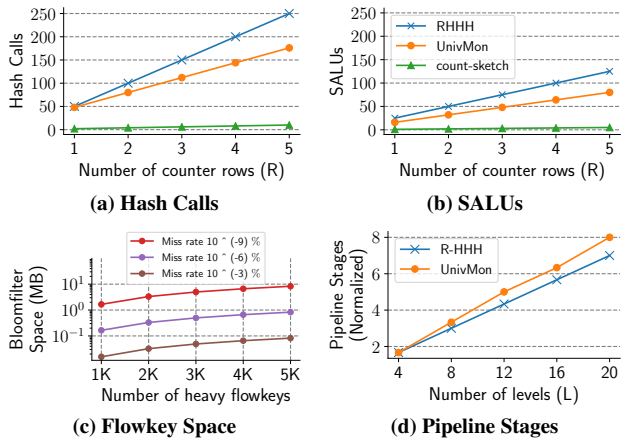
(a) Hash Calls

(b) SALUs

(c) Flowkey Space

(d) Pipeline Stages

Figure 6: Resource bottlenecks for sketch implementations.

of: $W = 2048$, $R = 3$, and $L = 16$ for UnivMon and $L = 25$ for R-HHH [12], which provides an error of up to 15% when processing packets from an inter-ISP packet trace [7]. We choose the value for L from the original papers [12, 41].

Fig. 6 illustrates how the use of four bottleneck resources depends on key sketch parameters. While the amount of available hardware resources can differ across hardware vendors and versions, we see that resource usage increases rapidly as we need more counters to meet higher accuracy requirements. While we cannot report exact resource usages due to proprietary reasons, we note that UnivMon and R-HHH are infeasible today on the hardware for many configurations. Perhaps more importantly, switches must also support tasks other than sketch-based telemetry (e.g., [33, 43]). Thus, it is critical to reduce the resource footprint of the sketches to ensure they can co-exist with other switch functions.

B1. Hash calls: Recall that count sketch needs $2R$ hash calls per packet (§2.1), matching the results in Fig. 6a. UnivMon and R-HHH execute one count sketch per level L. As a result, R-HHH needs $L \cdot 2R$ hash calls. UnivMon needs to compute an additional L 1-bit hash calls in its sampling stage, adding up to $L \cdot (2R + 1)$ hash calls.

At first glance, it may seem that the number of hash calls is not a bottleneck as these are called on demand per packet. While this is true in a software setting, where only the required calls are performed on demand, hashing on hardware is different. On a hardware switch, all hash calls appearing in the code need to be *pre-allocated* since execution at line rate must be guaranteed for all possible execution paths. This increases resource requirements, even if hash calls need not be executed. For example, even though UnivMon and R-HHH (Fig. 2) may not update all levels of count sketches for all packets, all hash resources must be pre-allocated.

B2. Memory accesses: Count Sketch maintains R counter arrays (§2.1) and for each row it must read one counter from memory and update its value. This means that count sketch needs R counter updates per packet, requiring R Stateful ALUs (SALUs) as shown in Fig. 6b. When the compiler compiles

the P4 code of UnivMon and R-HHH in Fig. 2, it allocates separate memory regions and SALUs for each level of count sketches, thus SALU requirements are proportional to the number of levels L. Since we need R memory processes per packet for the count sketch at each level, we need a total $L \cdot R$ SALUs for R-HHH and UnivMon. This makes memory access hardware (SALU) a bottleneck (Fig. 6b). Similar to hash resources, SALUs need to be *pre-allocated* at compile time, even if they may remain unused.

B3. Resources for tracking heavy flowkeys: Many sketches need to track heavy flowkeys to enable downstream analytics tasks. Typically, these sketches store heavy flowkeys in a separate data structure (e.g., heap or priority queue) [17, 41].

In practice, however, the exact details of if/how this can be realized on switch hardware are unclear. Specifically, a heap or priority queue, while feasible in software switches is too complex to be implemented on the programmable hardware switch. Alternatively, the data plane can relay all flowkeys to the switch control processor or record all flowkeys in the data plane. However, these are not feasible; e.g., bandwidth between the data plane and the control plane is limited, and data plane memory is also limited. Some sketch constructions store heavy flowkeys together with the counters [11, 32, 46]. However, these are infeasible at line-rate on today's RMT switches.[3]

To reduce the memory use, prior work proposed an optimized baseline—when a packet arrives, it checks whether the frequency of a flowkey has exceeded the threshold by querying the sketch counter, and if so, it reports the key to the control plane [33, 39, 41]. Unfortunately, this still has a problem as "heavy" flowkeys may be reported redundantly every time a packet arrives and needs more control plane bandwidth. To avoid duplicate reporting to the control plane, we could use a Bloom filter to check if a heavy key has already been reported [33]. However, we need to configure the Bloom filter (i.e., bitmap size and number of hash functions) to have really low false positives since a false positive in the Bloom filter for the duplicate check is a potential miss of a heavy flowkey. Fig. 6c confirms this trade-off; we can configure the Bloom filter depending on the target miss rate and we find the memory footprint is correspondingly higher (We use 3 hash functions for Fig. 6c). Using a Bloom filter might be a valid approach if we allow some missing heavy flowkeys, we argue a design that targets a zero miss rate is more desirable.

We implement four possible strawman solutions to report heavy flowkeys and run microbenchmarks on a Tofino hardware switch to understand a trade-off between the accuracy and the resource consumption. Table 1 summarizes our analysis and shows that we have an undesirable trade-off between the miss rate of heavy flowkeys, data plane resources (mem-

---

[3]Specifically, HashPipe [46] cannot be directly implemented on RMT architecture due to complex memory access patterns (see [11] for more details). Precision [11] requires recirculation, which means some packets must go through entire pipeline again.

	Miss rate	CP bandwidth	DP resource
Recored every key in the DP	Zero	Low	Infeasible
Report every key to the CP	Zero	Infeasible	Low
Report heavy keys to the CP	Zero	Infeasible	Low
Report non-duplicate heavy keys	Low	Low	High

Table 1: Strawman solutions for tracking heavy flowkeys (CP: control plane, DP: data plane).

ory for keys and hash calls for Bloom filters), and the control plane bandwidth (for reporting keys).

B4. Pipeline stages: So far we have implicitly assumed that the switch has a single pool of resources on the switch (i.e., SRAM/TCAM, SALUs, and hash calls) that can be allocated to the sketch operations. In reality, the resources are partitioned across the pipeline stages. This impacts resource use in two ways. First, before an operation can be assigned to a stage, all required resources need to be available on that stage. If that is not the case, it needs to be moved to the next stage. Second, if there is a dependency between two operations, e.g., $O_1 \rightarrow O_2$ in the code, then O_2 must be placed on a later stage than O_1, even if there are unused resources available on stages earlier in the pipeline. For example, the sequential if clauses used by UnivMon (Fig. 4) create sequential dependencies between the *if* clauses.

This means that, depending on resources required by operations and dependencies between them, the compiler will only be able to use a subset of the resources on the switch. To account for this, we consider pipeline stages as a separate resource. Fig. 6d shows the number of pipeline stages needed as a function of level L if we respect this architectural constraint. We see that UnivMon requires similar or more pipeline stages than R-HHH with same configuration parameters and the gap is increasing as the number of levels increases. This is a direct result of the sequential dependencies in UnivMon. The number of pipeline stages used is measured by running the RMT resource mapper.

4 Optimizations

Next, we present a series of optimizations to address the resource bottlenecks we identified earlier. For each optimization, we discuss the key idea, before discussing the correctness and applicability constraints. Some of these optimizations (e.g., O1, O3, O4) have appeared in earlier theoretical efforts and demonstrated in other settings (e.g., FPGA, software switch). Our contribution here is translating these ideas to hardware switches. Others (O2, O5, O6) are novel to the best of our knowledge. As summarized in Table 2, our optimizations can be applied to a broad spectrum of published sketches for telemetry and benefit 15 out of the 19 sketches listed. Some sketches that are outside our scope cannot be supported as they either use: (1) processing logic that is infeasible in hardware (i.e., Hashpipe); (2) counter data structures different

Sketch Type	Sketch Name	Feasible on HW?	Applicability of SketchLib
Frequency Estimation / Heavy Hitters	Count-Min [21]	Yes	O6
	Count Sketch [17]	Yes	O1, O6
	MRAC [37]	Yes	O3, O5
	Hashpipe [46]	No	N/A, due to infeasible logic
	Precision [11]	Yes	No, uses packet recirculation
Hierarchical Heavy Hitters	RHHH [12]	Yes	O1, O2, O5, O6
	HHH [20]	Yes	O1, O6
Cardinality	PCSA [25]	Yes	O3, O5
	MRB [23]	Yes	O3, O5
	LogLog [22]	Yes	O3
	HyperLogLog [26]	Yes	O3
Entropy	EntropySketch [38]	Yes	O1
Change Detection	K-ary [36]	Yes	O1, O2, O6
Super Spreaders	SpreadSketch	Yes	O3, O5
	BeauCoup [18]	Yes	No, non-counter based sketch
General	UnivMon [41]	Yes	O1, O2, O3, O4, O5, O6
	FCM [47]	Yes	O6
	SketchLearn [32]	Yes	O2
	ElasticSketch [53]	Yes	Not applicable

Table 2: Applicability of SketchLib on existing sketches.

from sketches (i.e., BeauCoup); or (3) complex processing patterns such as packet recirculation (i.e., Precision).

4.1 Optimizing Hash Calls

Both single- and multi-level sketches need to compute multiple hash functions, resulting in high hash call usage in the hardware pipeline. We describe two optimizations: consolidating short hash calls and reusing hash calls.

Optimization 1. Consolidate many short hash calls. We observe that many hash calls only need short-length (e.g., 1-bit) hash results. For instance, count sketch (Fig. 1) computes a series of 1 bit hash calls, s_1 to s_R. Similarly, UnivMon (Fig. 2 (b)) computes h_1 to h_L. We can reduce the number of hash calls by consolidating many short hash calls, as long as the inputs to the hash calls are the same.

Consider a count sketch with $R \times W = 3 \times 512$ counters. Per row, we need two hash results: a 9-bit (i.e., $\log_2 512 = 9$) hash to index into the counter array and a 1-bit hash for the "sign" of the counter. Instead of using $3 \cdot 2 = 6$ hash calls, we can instead use one hash call that returns a 30-bit result to provide the 6 hash calls as in Fig. 7. Note that splitting a long hash result only needs simple hardware shift and bit mask operations. R-HHH and UnivMon are also benefited as they use multiple count sketches. Further, UnivMon uses many 1-bit hash calls in its sampling stage.

Correctness and applicability: For this optimization to be valid, the split short-bit hash results from the longer hash result must use the same flowkey as the input and, if required by the sketching algorithm, be pairwise independent [17]. Independence is achieved by randomly picking (different) seeds for hash calls in practice [12, 41, 53]. Theoretical anal-
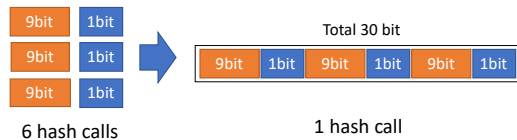
Figure 7: Optimization 1 reduces hash calls for count sketch.

Flowkey	Seed	Additional Condition	Opt
same	diff.	Sum of hash bit length is less than max capacity	O1
same	same	-	O2
diff.	same	One level of hash calls is executed	O2
diff.	diff.	-	-

Table 3: Conditions for optimization 1 and optimization 2.

ysis in other contexts [24, 35] shows that using different bits from the same hash call can also provide independence. Empirically, recent work [51] shows no accuracy loss for Univmon and our results (§6) confirm this with other sketches listed in Table 5. In addition, hash calls need to be short so that the sum of hash bit length is less than the length of one call (e.g., 32 bits). Fortunately, many single- and multi-level sketches [12, 17, 23, 25, 26, 37, 41] satisfy this condition.

Optimization 2: Reuse the hash calls across levels for multi-level sketches. Our second insight is that we can reuse the hash calls if there are no independence requirements across them ; i.e., they can use the same seed. Although hash independence is usually required across different counter arrays within a single level sketch, it is not required *across* levels [16]. Thus, we can use the *same* hash seed cross different levels for multi-level sketches.

Specifically, the original implementations of R-HHH and UnivMon (see Fig. 2) use a *different* hash seed in each of the *CS_level_i* count sketch executions. We can modify the code to reuse the same hash seed and reuse hash results when independence is not needed. This optimization reduces the number of hash calls significantly. For example in Fig. 2, R-HHH and UnivMon each have a set of hash calls F_i as $\{f_{i1}, f_{i2}, ... f_{i(2R)}\}$ at each level i of count sketch, resulting in $L \cdot 2R$ hash calls. By simply changing all of F_i to F_1, we reduce hash call usage from $L \cdot 2R$ to $2R$. For R-HHH, the result of F_1 is used to update one selected level of count sketch, and for UnivMon, result of F_1 can be used to update potentially multiple levels per packet.

Correctness and applicability: Reusing seed values across levels does not affect the theoretical independence requirements [16]. We empirically confirm in the evaluation that this optimization achieves similar accuracy (§6.1).

Table 3 summarizes the conditions under which the two hash optimizations are used. Note that for O2, if different levels' hashes have diverse output bit-length requirements, the hash call with the longest output bit-length will be used to supply hash results with various bit lengths. Also we need to make sure that the hash seeds are either the same in the first place or can be set to the same for O2 to apply.
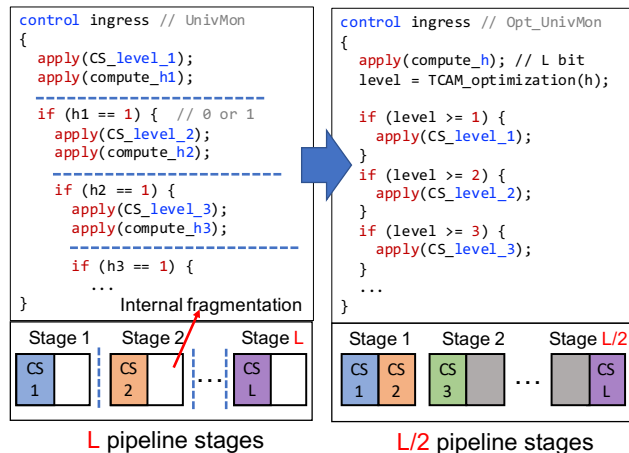
Figure 8: Optimization 3 removes the sequential computation dependency and reduces the usage of pipeline stages.

4.2 Optimizing Pipeline Stages

The sequential `if` clauses are observed in both single and multi-level sketches. This creates sequential compute dependencies and entails high usage of pipeline stages.

Optimization 3: Avoiding the sequential `if` clauses using a longest prefix match. To explain this optimization, we use UnivMon (Fig. 8) as an example. Deciding which levels to be updated for each flowkey creates a logical *dependency* between levels. Specifically, level i+1 needs to be updated only if the value of h_i returns 1 for hash functions $h_i : [n] \rightarrow \{0/1\}$. These L-level dependencies lead to an implementation as Fig. 8-left using sequential `if` clauses with hash values $(h_1, h_2, \ldots, h_L)$.

To address this bottleneck, our insight is that the number of leading 1-bits in $(h_1, h_2, \ldots, h_L)$ represents the sequence of "true" conditions in the `if` clauses. We observe that this is equivalent to the *longest prefix match* (LPM), which can be computed efficiently in hardware. That is, we can compute L hash bits together using a single L bit hash and use LPM to identify which layers need to be updated. This LPM operation is realized via TCAM as shown in Fig. 9. We insert rules with 1- and wildcard bits corresponding to each level and perform LPM to obtain the last level of UnivMon for each flowkey. LPM is relatively cheap—can be done in one pipeline stage using a small amount of TCAM. With this optimization, we can reduce the usage of pipeline stages by half if one count-sketch consumes half of the resources in one pipeline stage (Fig. 8-right).

Correctness and applicability: Our refactored implementation has the same functionality, resulting in the same updates to the sketch arrays. This optimization applies to many single and multi-level sketches that build on the power-of-two choices observation [23, 25, 26, 37, 55].
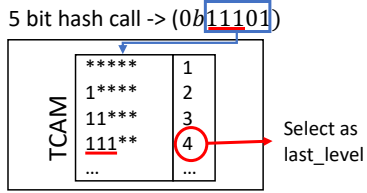
Figure 9: Replacing the sequential `if` clauses via TCAM.

4.3 Optimizing Memory Accesses

Sketches require memory accesses for their counter updates, leading to high SALU usage. This becomes especially significant for multi-level sketches.

Optimization 4: Refactor multi-level sketches to update one level per packet. We refactor multi-level sketching algorithms and their code to guarantee only one level is updated per flowkey. Recall that UnivMon needs to update one or more levels of count sketch (CS) for each packet with its flowkey. In Fig. 10 (top), a flowkey of packet K_{green} updates three levels, K_{gray} updates two levels, and K_{red} updates all levels of count sketch. Instead, our modified algorithm is guaranteed to update only the "last" level for each packet, as shown in Fig. 10 (bottom). The modified algorithm becomes structurally similar to other multi-level sketches that natively update only one level [12, 23, 25, 37]. As a result, the processing overhead is significantly reduced.

This "update-last-level" idea was also proposed to optimize UnivMon for embedded platforms [52] and software switches [40, 51]. Our contribution here is: (1) to extend this to programmable switches and (2) to generalize the idea to support updating arbitrary levels. Based on the algorithmic design, different multi-level sketches may require different optimization strategies to update a level (e.g., RHHH [12] modifies HHH [20] by randomly selecting a level to update). To implement this optimization, we can insert user-defined ternary rules in TCAM (as O3) to classify packets into different levels in a multi-level sketch.

Correctness and applicability: By construction, our modified algorithm provides equivalent functionality as the original version. As shown on the right side of Fig. 10 with K_{green} flowkey as an example, Levels 1 and 2 do not need to be updated anymore. Level 3 has the estimated flow count for this particular flow with the same or better accuracy since Level 3 only processes a smaller amount of traffic than Levels 1 and 2. Thus, the estimated count of K_{green} from Level 3 can be reused for Levels 1 and 2. This applies to all other flowkeys during the offline estimation in the network control plane.

To apply this optimization, a multi-level sketch should meet two conditions: (1) the original algorithm has multiple sketch updates per packet, and (2) it is algorithmically correct to reduce the multi-level updates to one per packet. That said, we acknowledge that there are scenarios where this optimization is not directly applicable. For instance, it is not obvious if/how we can refactor some multi-level sketches such as
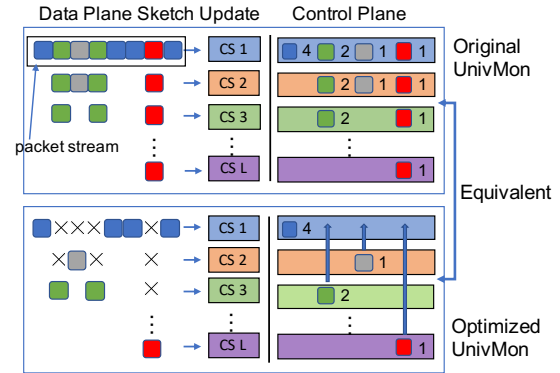
Figure 10: UnivMon updates only the last level per packet. CS stands for Count-Sketch.
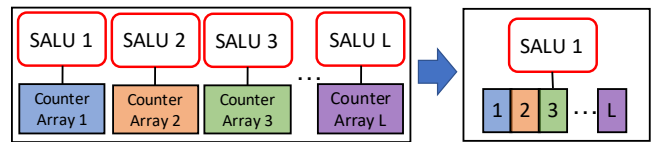
Figure 11: Optimization 5 removes unnecessary allocated SALUs by rewriting P4 code.

SketchLearn [32] to update only one level per flowkey (if possible). This requires future research.

Optimization 5: Remove unnecessary SALU operations. A multi-level sketch maintains multiple independent levels of sketches. For each counter at each level, the compiler statically allocates an SALU for memory access. This results in high SALU usage, even if only one level needs to be updated per packet; i.e., usage is the same as updating all levels.

We can remove unnecessary SALUs when only one update is needed per packet. The reason why the compiler inefficiently preallocates SALUs for all possible memory accesses is that it is difficult to automatically figure out that only one update is needed at runtime. Our optimization restructures the P4 code to make this explicit for the compiler that only one count sketch update is needed per level. Instead of using separate counter arrays located in different switch memory regions, we consolidate the counter arrays of all levels in a single array located in one region of memory. This is possible because SALU can support random access, thus based on the selected level, we can compute the corresponding index value to access the consolidated register. Fig. 11 illustrates this SALU optimization. This optimization reduces the SALU requirements for multi-level sketches by a factor of L (the number of levels, e.g., 25 for R-HHH [12]).

Correctness and applicability: This technique does not affect accuracy because the modified code has the same functionality as the original version. We can apply the optimization to multi-level sketches that have the property of updating only one level per flowkey. There are many multi-level sketches satisfying this property [12, 23, 25, 37, 41].

Resource Bottlenecks	Optimizations	API
Hash Calls	O1. Consolidate short-bit hash calls	`hash_consolidate_and_split()`
	O2. Reuse hash calls across levels	`select_key_and_hash()`
Pipeline Stages	O3. Remove sequential if clauses using TCAM	`tcam_optimization()`
SALUs	O4. Update only one level per flowkey	-
	O5. Rewrite P4 code to reduce memory accesses	`consolidate_update()`
Resources for tracking heavy flowkeys	O6. Use a hash table to remove duplicate flowkeys	`heavy_flowkey_storage()`

Table 4: The relationships among the bottlenecks, optimizations and API calls.

4.4 Optimizing Heavy Flowkey Reporting

Optimization 6: Use a hash table and an exact-match table for checking duplicate flowkeys. As discussed in §3.2 B3, prior efforts [33, 39] use Bloom filters as the duplicate checker but the false positives from the filters will cause misses of heavy flowkeys, unless a very large Bloom filter is used. To improve this tradeoff between miss rate and data plane resource, we use a hash table and an exact-match table to check duplicates. Specifically, the hash table stores heavy flowkeys and detects whether there is a collision. For each heavy flowkey, if it is already stored in the hash table or exact-match table, it will not be reported to the controller; otherwise, it will be inserted to the hash table. But if this flowkey collides with another key in the hash table, then it will be reported to the controller which then inserts this flowkey to the exact-match table to filter future duplicate keys. In this way, we can ensure a zero miss rate on reporting heavy flowkeys.

Correctness and applicability: This optimization ensures a zero miss rate of heavy flowkeys because when collisions happen in the hash table, the flowkeys are reported to the control plane and inserted to the exact-match table (as a secondary duplicate checker). No unique heavy flowkeys are dropped in this mechanism. Compared to Bloom filters, this approach adds some additional control plane bandwidth when collisions happen in the hash table. As we evaluate in §6.5, this added bandwidth is small (e.g., 2% increase). This optimization can be applied to both single- and multi-level sketches requiring heavy flowkey tracking [12, 17, 41].

5 SketchLib API

In this section, we present our P4 API for helping sketch developers to use our optimizations. For each API call, we show the implementation for the macro and how the macro is used. SketchLib API supports both P4-14 and P4-16 [6]. Table 4 maps the optimizations to the API calls.

`hash_consolidate_and_split(Key,Seed,List(BitLen),BL_sum,List(Mask))`[4] reduces hash calls by consolidating small bit hash calls (O1). Fig. 12 shows how a sequence of short hash calls is replaced by a macro that uses only a single hash call with length the sum of all

BitLen of the shorter hashes. The resulting hash value is then partitioned in shorter hashes. For P4-14, we split the result using `modify_field_with_shift(dst, src, shift, mask)` primitive (i.e., `dst = (src » shift) & mask`) where `mask` is a series of 1's with `BitLen` as shown. For P4-16, the same principle is applied, but bit slice operation (e.g., `h[BL1:0]`) is used. Note that the macro specifies both the number of short hashes being merged (*List*) and the names of the short hashes, so multiple macros must be defined if O1 is applied multiple times.

```
1: h1 = hash(sIP, seed1, 5);
2: h2 = hash(sIP, seed2, 3);
3: h3 = hash(sIP, seed3, 4);
        ↓
1: hash_consolidate_and_split_3
    (sIP, seed1, 5, 3, 4, 12,
    0b11111, 0b111, 0b1111)
```

```
1:#define
    hash_consolidate_and_split_3
    (Key, Seed, BL1, BL2, BL3,
    BL_sum, mask1, mask2, mask3)
2: h = hash(Key, Seed, BL_sum);
3: h1 = h & mask1;
4: h2 = (h >> BL1) & mask2;
5: h3 = (h >> (BL1+BL2)) & mask3;
```

Figure 12: `hash_consolidate_and_split()`

`select_key_and_hash(List(Key),Level,Seed,BitLen)` implements O2 for the case one of the several hash calls with different `Key` and same `Seed` is selected for execution. Here, we can select the key in advance and use only one hash call to get the result as in Fig. 13. For instance, R-HHH can be optimized by using this API call. The example shown is a single hash call, but if multiple are needed (e.g. sketch with *R* =5 needs 5 hash calls), the number of hash calls can be increased. For the sketches that share the same `Key` and `Seed` (e.g., UnivMon), no separate API call is necessary since the hash value can simply be reused.

```
1: if (V == 1)
2:   h = hash(key1, seed, 3);
3: if (V == 2)
4:   h = hash(key2, seed, 3);
5: if (V == 3)
6:   h = hash(key3, seed, 3);
        ↓
1: select_key_and_hash_3
    (key1, key2, key3, V, seed, 3)
```

```
1: #define select_key_and_hash_3
    (key1, key2, key3, V, Seed, BL)
2: if (V == 1)
3:   k = key1;
4: if (V == 2)
5:   k = key2;
6: if (V == 3)
7:   k = key3;
8: h = hash(k, Seed, BL);
```

Figure 13: `select_key_and_hash()`

`tcam_optimization(Hash_Result)` implements O3 to remove sequential `if` clauses by applying an equivalent a LPM table which uses TCAM to which levels need to be updated. The macro implements the use of the TCAM to look up the level (see Fig. 8).

`consolidate_update(Level,Index)` implements O5 to reduce memory accesses, as illustrated in Fig. 14. `Level`

---

[4]While there is no concept of `List` in P4, we use it to describe the type of parameters conceptually throughout this section. In our API implementations, it is converted to multiple parameters; e.g., `List(BitLen)` → `(BL1, BL2, BL3)` as shown in Fig. 12.

indicates the selected counter array and `Index` references the location for the memory update within the counter array. The API call consolidates counter arrays and computes the new address for the consolidated array. `size` indicates the bit length (e.g., 10) of the width (e.g., 1024).
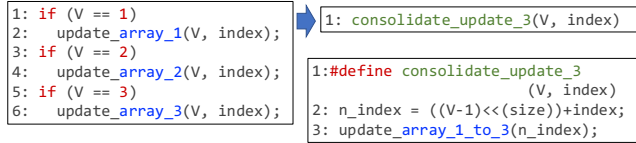
```
1: if (V == 1)
2:   update_array_1(V, index);
3: if (V == 2)
4:   update_array_2(V, index);
5: if (V == 3)
6:   update_array_3(V, index);
```
⟶
```
1: consolidate_update_3(V, index)

1:#define consolidate_update_3
                        (V, index)
2: n_index = ((V-1)<<(size))+index;
3: update_array_1_to_3(n_index);
```

Figure 14: `consolidate_memory_update()`

`heavy_flowkey_storage(Key,List(Estimate),Threshold)` reduces the memory space for heavy flowkeys (O6). The challenge is checking whether the estimated flow count is above a threshold entirely in the data plane. Specifically, this entails computing the median value based on an estimated flow count from each row and comparing it to the threshold value. However, computing the median is not supported in the data plane. Instead, we leverage the fact that we can check whether the median of a set of values exceeds a threshold without computing the median as follows. We compare all of estimated flow count for all rows, as shown in lines 3-9 in Fig. 15 which is for R = 3 case. Then, the condition (sum (s1, s2, s3) $\geq$ 2) at line 11 is equal to (median(est1, est2, est3) > T).[5] This can be generalized for different Rs. We implement the duplicate filter using a hash table and a exact-match table. If a flowkey collides with an entry in the hash table and the exact-match table does not have an entry for the flowkey, we report it to switch control plane via a PCIe channel. Upon receiving the reported key, the switch control plane CPU adds entries into the exact-match table.

6 Evaluation

In this section, we evaluate the benefits of SketchLib on seven sketches. Across a range of settings, we see that SketchLib can reduce the resource footprint of sketches on switch hardware (up to 96%) while achieving similar accuracy.

6.1 Experimental Setup

Sketches. We implement all 15 sketches in Table 2 using SketchLib and source codes for sketches are available at [6]. Among 15 sketches, we pick seven representative sketches for our evaluation as in Table 5.

Testbed. We evaluate SketchLib on a local testbed with an Edgecore Wedge 100BF Tofino-based programmable switch and a server equipped with dual Intel Xeon Silver 4110 CPUs, 128GB RAM, and a 100Gbps Mellanox CX-4 NIC connected to the switch. We use the P4-16 version of SketchLib with Tofino SDE version of 9.1.1 in our experiments.

---

[5]For Count-Min sketch [21], we can use (sum (s1, s2, s3) $\geq$ 1).

```
01:#define heavy_flowkey_storage_3
                    (Key, Est1, Est2, Est3, T)
02:
03: s1, s2, s3 = 0;
04: if (Est1 > T)
05:   s1 = 1;
06: if (Est2 > T)
07:   s2 = 1;
08: if (Est3 > T)
09:   s3 = 1;
10:
// above threshold test
11: if (s1 + s2 + s3 >= 2) {
12:   if (HT[h(Key)] == empty) { // HashTable
13:     HT[h(Key)] = Key;
14:     send_to_cpu(Key);
15:   } else if(HT[h(Key)] != Key) {
16:     if (!(flowkey in MT)) { // MatchTable
17:       send_to_cpu(Key);
18:     }
19:   }
20: }
```

Figure 15: `heavy_flowkey_storage()`

Traces. We use five CAIDA backbone traces capture at 3/20/14 to 6/19/14 Sanjose, 1/21/16 Chicago, 5/17/18 to 8/16/18 New York City [7]. We split one hour traces into 30 second epochs. Each epoch includes about 12M-23M packets, with 398K distinct source IPs, 280K distinct destination IPs, and 1.6M distinct 5 tuples.

	Level (L)	Row (R)	Width (W)	Space
CS [17]	-	5	4096	80KB
HLL [26]	-	-	2048	8KB
UnivMon [41]	16	5	2048	640KB
R-HHH [12]	25	3	2048	600KB
MRAC [37]	12	-	2048	96KB
MRB [23]	16	-	4096	8KB
PCSA [25]	32	-	20	0.125KB

Table 5: Sketch parameters for evaluation.

Sketch parameters. Table 5 shows the configuration parameters for the sketches. Most sketches use 4 byte counters. The cardinality estimators (e.g., MRB and PCSA) use bitmap thus each counter is 1 bit.

Metrics. Depending on the sketch and the measurement task, we report two error metrics. For each metric, we run the experiment 5 times independently with different hash parameters and report the 25%, 50%, 75% percentiles of the errors. For brevity, we report results using source IP as the flowkey except for R-HHH, noting that the results are qualitatively similar for other types of flowkeys. R-HHH uses (source IP, destination IP) pair as flowkey as presented in the original paper [12].

- Average Relative Error (ARE): $\frac{1}{k}\sum_{i=1}^{k}\frac{|f_i-\hat{f}_i|}{f_i}$, where k means the top k heavy flows. f_i is actual flow count for flow i and $\hat{f}_i$ is the estimated flow count from the sketch. $f_i \geq f_{i+1}$ for any i, thus it is sorted in descending order. We use k=50 for count sketch and R-HHH.

- Relative Error (RE): $\frac{|True-Estimate|}{True}$, where *True* is ground truth value and *Estimate* is estimated value. We use this metric for sketches that estimate cardinality and/or entropy.
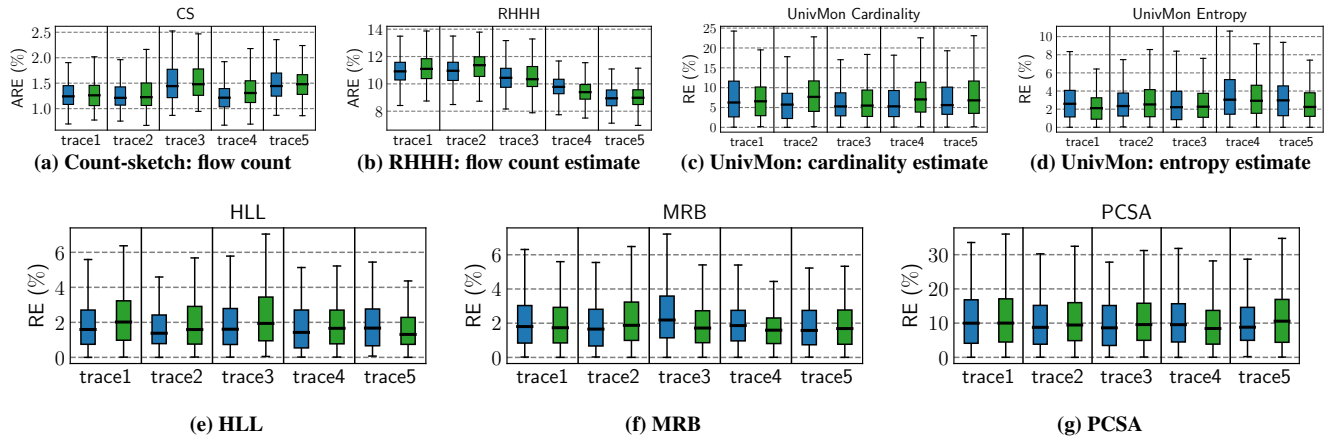
Figure 16: Accuracy comparison of sketches between original and optimized sketches across traces. Left: original, Right: optimized.
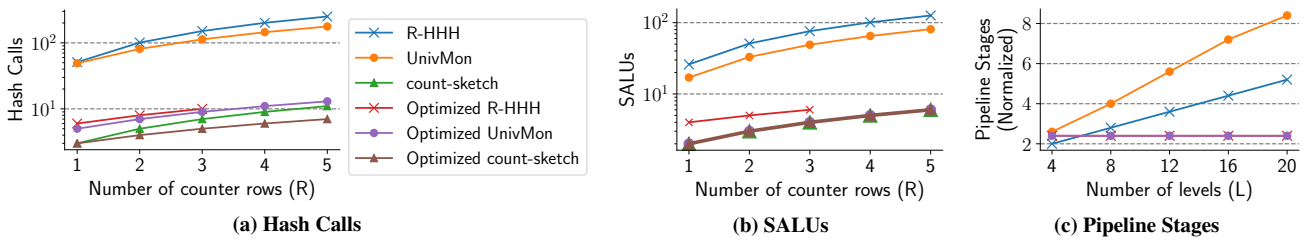

Figure 17: Resource consumption before/after optimizations.

6.2 Accuracy

We run the accuracy experiment of SketchLib in two ways. First, we show the accuracy is preserved between baseline software implementation and hardware implementation with SketchLib (§6.2.1). Second, we compare the accuracy of the hardware implementations with and without SketchLib (§6.2.2).

6.2.1 Comparison with the Software Baseline

Reporting methodology. We compare the accuracy of the sketch refactored with SketchLib (on hardware) against a baseline software implementation. The baseline software implementation runs sketches on the software. We run experiments over multiple traces with independent runs. After optimizing sketches with SketchLib, we run experiments on Tofino hardware with all five traces. For each one-hour trace, we randomly sample 40 30-second epochs and obtain 5 accuracy numbers per epoch with independent trials. The server replays traces to the switch using tcpreplay at a speed of 800K packets/second. Between epochs, we wait for switch control plane to pull counters and flowkeys from the data plane (see §7).

Result. Fig. 16 empirically validates that SketchLib optimizations achieve similar accuracy. For every trace, the left blue bar represents the software baseline and the right green bar is the hardware reported result with SketchLib applied.[6] Fig. 16a - Fig. 16d shows the accuracy of sketches that need to track heavy flowkeys and the rest show sketches that need

---

[6]We do not show MRAC as the estimation logic for MRAC is not public.

UivMon	Without SketchLib			With SketchLib
Level (L)	8	6	5	16
Row (R)	4	5	6	5
Width (W)	32768	32768	32768	2048
RE (%)	95.4%	98.8%	99.4%	9.5%

Table 6: Relative error in cardinality estimation with and without SketchLib.

to maintain only counter arrays. Fig. 16c and Fig. 16d show the errors of UnivMon for cardinality estimation and entropy estimation. We can visually confirm that the distributions of accuracy before and after optimizations are similar.

6.2.2 Accuracy Improvement with SketchLib

Reporting methodology. We want to compare the best accuracy between with and without SketchLib on the hardware. We use UnivMon for this experiment. To systematically sweep configuration parameters for the best accuracy without SketchLib, we exploit the property of UnivMon. Among three sketch parameters level (L), row (R), and width (W), L is the most critical parameter, thus we pick three highest feasible L. Then we find maximum R and lastly W. Given fixed L, we explored different parameter R other than maximum R but result was similar. We use the simulator with 40 samples of trace1. With SketchLib, we use the same configuration from the original UnivMon paper.

Result. Table 6 shows that all feasible configurations without SketchLib show high error rate more than 95%. On the other hand, UnivMon with SketchLib shows low error rate of 9.5%.

Sketches	Hash Calls (O1/O2)	SALUs	Pipeline Stages
CS	31%/0		9%
HLL	80%/0		86%
UnivMon	44%/47%	90%	65%
RHHH	32%/60%	92%	62%
MRAC	87%/0	91%	68%
MRB	90%/0	93%	76%
PCSA	92%/0	96%	86%

Table 7: Individual resource reductions by optimizations.

6.3 Switch Resource Consumption

Next, we report the resource usage improvements on the identified resource bottlenecks (Table 7). The sketch parameters used are reported in Table 5.

Reporting methodology. We measure resource usages from original implementation using RMT resource mapper and optimized implementation using Tofino compiler to measure resource reductions. To factor out resource reductions for different optimizations in Table 7, we wrote P4 code with individual optimizations applied using SketchLib APIs to measure the resource usages.

Hash calls. Table 7 shows that using O1 to consolidate the 1-bit hash calls is effective for both single and multi-level sketches. For example, the number of hash calls for count sketch is reduced by 31%. R-HHH and UnivMon benefit from O1 as they are composed of multiple count sketches. Further, PCSA, MRAC, MRB and HLL have a series of 1-bit hash calls which O1 improves. For UnivMon and R-HHH, we can apply both O1 and O2 by reusing hash calls across levels to further reduce hash calls by over 90%. We further investigate the sensitivity of the reduction of hash calls vs. sketch parameters in Fig. 17a.[7] Multi-level sketches UnivMon and R-HHH have significant reduction and the resource used is close to single-level count sketch.

Stateful ALUs. O5 applies only to multi-level sketches and reduces SALU usage significantly if there are many levels in the sketch. With 16-32 levels, O5 saves 92% to 96% of the SALUs. We can see in Fig. 17a that O5 reduces SALU of UnivMon and R-HHH significantly across rows.

Counter Memory Space. Interestingly, O5, which we designed to reduce SALU usage, additionally reduces memory space. Investigating this further, we find that original sketch implementations have a memory region fragmentation problem. One counter array is smaller than a block of SRAM, causing additional (unused) memory overhead per each counter array. O5 has the added benefit of consolidating counter arrays and achieve 54%–96% of resource reduction in memory space for multi-level sketches (not shown).

Pipeline stages. The reduction of pipeline stages depends on a combination of factors — hash calls, SALUs, and code dependencies. Table 7 shows reduced pipeline stages from 9% to 86% across sketches. Sketches where O5 applies (HLL, UnivMon, MRAC, MRB, PCSA) have a large reduction because it removes many sequential `if` clauses. Effectively, our

---

	FCM native	SketchLib-optimized		
Resource	FCM+topK	FCM(+O6)	CM	UnivMon
Pipe. Stage	8	8	7	12
SRAM	9.5%	10.8%	8.0%	7.3%
TCAM	0%	0%	0%	0.3%
SALUs	20.8%	14.6%	14.6%	12.5%
Hash Calls	13.9%	9.7%	11.1%	18.1%
Hash Bits	5.6%	4.0%	4.0%	4.9%

Table 8: Comparison of hardware resource utilization.

		SketchLib-optimized		
	FCM+topK	FCM(+O6)	CM	UnivMon
HH (ARE)	1.41%	0.01%	0.13%	0.73%

Table 9: ARE of heavy hitter detection.

# of flows	500K	1M	5M	10M	30M
FCM+topK	0.35%	0.84%	3.60%	6.15%	17.0%
SketchLib UnivMon	2.59%	2.08%	2.21%	2.36%	2.96%

Table 10: Entropy error (RE), FCM vs. SketchLib-optimized UnivMon.

optimization can make the footprint of multi-level sketches agnostic to number of levels (Fig. 17c).

6.4 Comparison with FCM

FCM [47] is a recently published sketch with general capability, and it is feasible on the programmable switch. Thus, we compare FCM against sketches optimized with SketchLib in terms of resource usages and accuracy. Table 8 shows resource utilization comparison between FCM and SketchLib optimized sketches. We use the same configuration from public FCM code [3], and make SketchLib-optimized sketch use similar resources to FCM.

Heavy hitter detection. Table 9 shows the accuracy result of heavy hitter detection. We can see that FCM+topK suffers from a high error rate because of an inefficient mechanism for tracking heavy flowkey (approximate topK implementation of ElasticSketch [53]). Note that if FCM deploys one of our optimizations for tracking heavy flowkeys, FCM+O6 reduces the error rate significantly from 1.41% to 0.01%. We use the simulator with 40 samples of trace1 and report median ARE.

Entropy and cardinality. Table 10 and Table 11 compare entropy and cardinality estimation accuracy between FCM+topK and SketchLib-optimized UnivMon. In the experiments, UnivMon reports top-200 heavy hitters per level. For entropy, UnivMon shows a relatively stable error rate (2∼3%) across workloads, whereas FCM is dependent on workloads and the error rate can go up to 17%. For cardinality, the error rate of UnivMon is moderately increasing [8], whereas FCM suddenly becomes unusable after 5M flows. This is because Linear Counting [50] is used to estimate cardinality in FCM.

6.5 Tracking Heavy Flowkeys

To evaluate the impact of O6, we consider three metrics: miss rate, control plane bandwidth, and data plane memory. We

---

[7]Missing point for R-HHH in Fig. 17 means it is infeasible.

[8]We observe that, when UnivMon reports more heavy hitters per level, the cardinality error rate decreases (e.g., 17.58% in 10M flows with top-1000).

# of flows	500K	1M	5M	10M	30M
FCM+topK	0.004%	0.107%	0.519%	100%	100%
SketchLib UnivMon	21.9%	20.7%	31.7%	39.5%	73.8%

Table 11: Cardinality error (RE), FCM vs. SketchLib-optimized UnivMon.

Resource	UnivMon	With SketchLib UnivMon + NFs (L2, L3, LB, FW)
Pipe. Stage	12	12
SRAM	7.3%	38.6%
TCAM	0.3%	25.0%
SALUs	12.5%	12.5%
Hash Calls	18.1%	18.1%
Hash Bits	4.9%	11.2%

Table 12: Sketches are infeasible without SketchLib. With SketchLib, there are rooms for additional network functions (L2/L3 forwarding, L4 load balancer, and stateful firewall).

compare SketchLib-optimized approach vs. an "optimal" software solution. For this evaluation, we use two sketches (CS, UM) that track "heavy" flowkeys. For each 1-hr trace, we split it into epochs as before, and set a target threshold corresponding to the top 0.2 percentile of flow sizes (The results are independent of the threshold; this is to make the experiment concrete). Across different traces and sketches, SketchLib incurs zero miss rate, and at most 2% increase in control plane bandwidth (due to small number of duplicates), using less than 400KB of data plane memory overall (independent of the threshold, results not shown for brevity). To put this in context, a Bloom-filter based strawman for suppressing duplicates as discussed in §3 configured with the same memory use has a miss rate of 0.2%. Overall, this confirms that SketchLib offers a more practical alternative to the infeasible, inaccurate, and/or expensive strawman solutions from §3.

6.6 Other Benchmarks

Additional Network Functions. After optimized with SketchLib, sketches can even coexist with additional network functions such as L2/L3 forwarding, L4 load balancer, and stateful firewall. Table 12 shows resource utilization for additional network functions.

Code simplification. In addition to the resource efficiency benefits, our optimizations also simplify the sketch implementations by reducing the lines of code, as shown in Table 13.

Compilation time. We also measured compilation time to see whether our modified code will add significant overhead to the compiler. We measure compile time is measured on the server specified in (§6.1). For most cases, there was a negligible (≤ 1 second) increase (not shown).

7 Related Work

Programmable switches. The programmable switch architecture was introduced by Bosshart et al [15]. Subsequent work proposed a programming framework [14], functional hardware [1], and also compilation workflows [34]. Other

Sketch	CS	HLL	UM	RHHH	MRAC	MRB	PCSA
Before	201	290	460	471	261	317	305
After	131	112	127	128	91	94	93

Table 13: Lines of code simplification (UM stands for UnivMon).

vendors have developed programmable pipelined architectures and compilation workflows from P4 or P4-like primitives [4, 5]. While our focus is on Tofino, our approach could be useful for other platforms as well.

Optimizing sketches. HashPipe [46] focused on heavy hitter detection, but is not feasible in the current hardware. Other work has focused on the optimizing sketching algorithms in software switches (e.g., [31, 40, 51]). However, some of their ideas do not translate into a hardware context. For instance, NitroSketch increases the memory footprint to reduce CPU consumption, but the key bottleneck in hardware is different. Similarly, other approaches split a sketch into a fast and slow path on the software switch (e.g., [31]). Unfortunately, this is not relevant in hardware since we need all operations to be in the fast path. Some recent work [51, 52] specifically focus on optimizing UnivMon for embedded platforms and software switches. We translate these insights to a switch hardware realization, and generalize beyond UnivMon.

Control plane reporting. While this work focuses on optimizing data plane components of sketch-based monitoring, there are other challenges in accurately retrieving sketch counters in the control plane. Naïvely retrieving the counters using the existing control plane APIs can result in poor accuracy due to a nonnegligible amount of read and reset delays. We analyze this problem and suggest recommendations in parallel work [44].

Other work in network telemetry. Our focus in this paper is on sketch-based telemetry. There are other efforts for complementary monitoring capabilities (e.g., [29, 30, 48]) and performance-oriented objectives (e.g., [28, 45]).

8 Conclusions

Given increasing traffic rates and rich telemetry required, we see the confluence of two trends: the use of sketching algorithms and programmable switch hardware. Unfortunately, existing sketch implementations are not efficiently realizable, thereby limiting their effectiveness and coexistence with other switch functions. To this end, we systematically analyze the resource bottlenecks, suggest correct-by-construction optimizations, and design a practical library to help developers use these optimizations. Our evaluations show that the SketchLib library is broadly applicable to many sketches and reduces their resource footprint while achieving similar accuracy.

This work focuses on a single sketch-based monitoring task written using SketchLib APIs. We plan to support multiple tasks on a switch and automate the optimizations by integrating our techniques with a compiler as future work.

References

[1] Barefoot Tofino. https://barefootnetworks.com/products/brief-tofino/.

[2] Broadcom Trident 3. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56870-series/.

[3] FCM-sketch source code. https://github.com/fcm-project/fcm_p4.

[4] Marvell LiquidIO SmartNICs. https://www.marvell.com/products/ethernet-adapters-and-controllers.html.

[5] Netronome Agilio SmartNICs. https://www.netronome.com/products/nfe/.

[6] Open Sourced SketchLib. https://github.com/SketchLib.

[7] The CAIDA UCSD Anonymized Internet Traces. https://www.caida.org/data/passive/passive_dataset.xml.

[8] P4$_{14}$ Language Specification. https://p4.org/p4-spec/p4-14/v1.0.5/tex/p4.pdf, 2018.

[9] NPL Specifications . https://nplang.org/npl/specifications/, 2020.

[10] ALIZADEH, M., EDSALL, T., DHARMAPURIKAR, S., VAIDYANATHAN, R., CHU, K., FINGERHUT, A., LAM, V. T., MATUS, F., PAN, R., YADAV, N., ET AL. Conga: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (2014), pp. 503–514.

[11] BEN-BASAT, R., CHEN, X., EINZIGER, G., AND ROTTENSTREICH, O. Efficient measurement on programmable switches using probabilistic recirculation. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)* (2018), IEEE, pp. 313–323.

[12] BEN BASAT, R., EINZIGER, G., FRIEDMAN, R., LUIZELLI, M. C., AND WAISBARD, E. Constant time updates in hierarchical heavy hitters. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017), pp. 127–140.

[13] BENSON, T., ANAND, A., AKELLA, A., AND ZHANG, M. Microte: Fine grained traffic engineering for data centers. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies* (2011), pp. 1–12.

[14] BOSSHART, P., DALY, D., GIBB, G., IZZARD, M., MCKEOWN, N., REXFORD, J., SCHLESINGER, C., TALAYCO, D., VAHDAT, A., VARGHESE, G., AND WALKER, D. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.* (2014).

[15] BOSSHART, P., GIBB, G., KIM, H.-S., VARGHESE, G., MCKEOWN, N., IZZARD, M., MUJICA, F., AND HOROWITZ, M. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. *ACM SIGCOMM Computer Communication Review 43*, 4 (2013), 99–110.

[16] BRAVERMAN, V., AND OSTROVSKY, R. Zero-one frequency laws. In *Proc. of STOC* (2010).

[17] CHARIKAR, M., CHEN, K., AND FARACH-COLTON, M. Finding frequent items in data streams. In *International Colloquium on Automata, Languages, and Programming* (2002), Springer, pp. 693–703.

[18] CHEN, X., LANDAU-FEIBISH, S., BRAVERMAN, M., AND REXFORD, J. Beaucoup: Answering many network traffic queries, one memory update at a time. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication* (2020), pp. 226–239.

[19] CLAISE, B. Cisco systems NetFlow services export version 9. RFC 3954.

[20] CORMODE, G., KORN, F., MUTHUKRISHNAN, S., AND SRIVASTAVA, D. Finding hierarchical heavy hitters in data streams. In *Proceedings 2003 VLDB Conference* (2003), Elsevier, pp. 464–475.

[21] CORMODE, G., AND MUTHUKRISHNAN, S. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms 55*, 1 (2005), 58–75.

[22] DURAND, M., AND FLAJOLET, P. Loglog counting of large cardinalities. In *European Symposium on Algorithms* (2003), Springer, pp. 605–617.

[23] ESTAN, C., VARGHESE, G., AND FISK, M. Bitmap algorithms for counting active flows on high speed links. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement* (2003), pp. 153–166.

[24] FLAJOLET, P., AND MARTIN, G. N. Probabilistic counting algorithms for data base applications. *Journal of computer and system sciences 31*, 2 (1985), 182–209.

[25] FLAJOLET, P., AND MARTIN, G. N. Probabilistic counting algorithms for data base applications. *Journal of computer and system sciences 31*, 2 (1985), 182–209.

[26] FLAJOLET, P., RIC FUSY, GANDOUET, O., AND ET AL. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *AOFA* (2007).

[27] GARCIA-TEODORO, P., DIAZ-VERDEJO, J., MACIÁ-FERNÁNDEZ, G., AND VÁZQUEZ, E. Anomaly-based network intrusion detection: Techniques, systems and challenges. *computers & security 28*, 1-2 (2009), 18–28.

[28] GHASEMI, M., BENSON, T., AND REXFORD, J. Dapper: Data plane performance diagnosis of tcp. In *Proceedings of the Symposium on SDN Research* (2017), pp. 61–74.

[29] GUPTA, A., HARRISON, R., CANINI, M., FEAMSTER, N., REXFORD, J., AND WILLINGER, W. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (2018), pp. 357–371.

[30] HARRISON, R., CAI, Q., GUPTA, A., AND REXFORD, J. Network-wide heavy hitter detection with commodity switches. In *Proceedings of the Symposium on SDN Research* (2018), pp. 1–7.

[31] HUANG, Q., JIN, X., LEE, P. P., LI, R., TANG, L., CHEN, Y.-C., AND ZHANG, G. Sketchvisor: Robust network measurement for software packet processing. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017), pp. 113–126.

[32] HUANG, Q., LEE, P. P., AND BAO, Y. Sketchlearn: Relieving user burdens in approximate measurement with automated statistical inference. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (2018), pp. 576–590.

[33] JIN, X., LI, X., ZHANG, H., SOULÉ, R., LEE, J., FOSTER, N., KIM, C., AND STOICA, I. Netcache: Balancing key-value stores with fast in-network caching. In *Proc. of ACM SOSP* (2017).

[34] JOSE, L., YAN, L., VARGHESE, G., AND MCKEOWN, N. Compiling packet programs to reconfigurable switches. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)* (2015), pp. 103–115.

[35] KIRSCH, A., AND MITZENMACHER, M. Less hashing, same performance: building a better bloom filter. In *European Symposium on Algorithms* (2006), Springer, pp. 456–467.

[36] KRISHNAMURTHY, B., SEN, S., ZHANG, Y., AND CHEN, Y. Sketch-based change detection: methods, evaluation, and applications. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement* (2003), pp. 234–247.

[37] KUMAR, A., SUNG, M., XU, J., AND WANG, J. Data streaming algorithms for efficient and accurate estimation of flow size distribution. *ACM SIGMETRICS Performance Evaluation Review 32*, 1 (2004), 177–188.

[38] LALL, A., SEKAR, V., OGIHARA, M., XU, J., AND ZHANG, H. Data streaming algorithms for estimating entropy of network traffic. *ACM SIGMETRICS Performance Evaluation Review* (2006).

[39] LIU, Z., BAI, Z., LIU, Z., LI, X., KIM, C., BRAVERMAN, V., JIN, X., AND STOICA, I. Distcache: Provable load balancing for large-scale storage systems with distributed caching. In *Proc. of USENIX FAST* (2019).

[40] LIU, Z., BEN-BASAT, R., EINZIGER, G., KASSNER, Y., BRAVERMAN, V., FRIEDMAN, R., AND SEKAR, V. Nitrosketch: Robust and general sketch-based monitoring in software switches. In *Proceedings of the ACM Special Interest Group on Data Communication*. 2019, pp. 334–350.

[41] LIU, Z., MANOUSIS, A., VORSANGER, G., SEKAR, V., AND BRAVERMAN, V. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference* (2016), pp. 101–114.

[42] METWALLY, A., AGRAWAL, D., AND EL ABBADI, A. Efficient computation of frequent and top-k elements in data streams. In *International Conference on Database Theory* (2005), Springer, pp. 398–412.

[43] MIAO, R., ZENG, H., KIM, C., LEE, J., AND YU, M. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017), pp. 15–28.

[44] NAMKUNG, H., KIM, D., LIU, Z., SEKAR, V., AND STEENKISTE, P. Telemetry retrieval inaccuracy in programmable switches: Analysis and recommendations. In *Proceedings of the Symposium on SDN Research* (2021).

[45] NARAYANA, S., SIVARAMAN, A., NATHAN, V., GOYAL, P., ARUN, V., ALIZADEH, M., JEYAKUMAR, V., AND KIM, C. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017), pp. 85–98.

[46] SIVARAMAN, V., NARAYANA, S., ROTTENSTREICH, O., MUTHUKRISHNAN, S., AND REXFORD, J. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research* (2017), pp. 164–176.

[47] SONG, C. H., KANNAN, P. G., LOW, B. K. H., AND CHAN, M. C. Fcm-sketch: generic network measurements with data plane support. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies* (2020), pp. 78–92.

[48] TAMMANA, P., AGARWAL, R., AND LEE, M. Distributed network monitoring and debugging with switchpointer. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)* (2018), pp. 453–456.

[49] WANG, M., LI, B., AND LI, Z. sflow: Towards resource-efficient and agile service federation in service overlay networks. In *Proc. of IEEE ICDCS* (2004).

[50] WHANG, K.-Y., VANDER-ZANDEN, B. T., AND TAYLOR, H. M. A linear-time probabilistic counting algorithm for database applications. *ACM Transactions on Database Systems (TODS) 15*, 2 (1990), 208–229.

[51] XIAO, Q., TANG, Z., AND CHEN, S. Universal online sketch for tracking heavy hitters and estimating moments of data streams. In *IEEE INFOCOM* (2020).

[52] YANG, M., ZHANG, J., GADRE, A., LIU, Z., KUMAR, S., AND SEKAR, V. Joltik: enabling energy-efficient" future-proof" analytics on low-power wide-area networks. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking* (2020), pp. 1–14.

[53] YANG, T., JIANG, J., LIU, P., HUANG, Q., GONG, J., ZHOU, Y., MIAO, R., LI, X., AND UHLIG, S. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (2018), pp. 561–575.

[54] YU, D., ZHU, Y., ARZANI, B., FONSECA, R., ZHANG, T., DENG, K., AND YUAN, L. dshark: a general, easy to program and scalable framework for analyzing in-network packet traces. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)* (2019), pp. 207–220.

[55] YU, M., JOSE, L., AND MIAO, R. Software defined traffic measurement with opensketch. In *Proc. of USENIX NSDI* (2013).

[56] ZHOU, Y., ZHANG, D., GAO, K., SUN, C., CAO, J., WANG, Y., XU, M., AND WU, J. Newton: intent-driven network traffic monitoring. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies* (2020), pp. 295–308.

A   Comparison of RMT resource mapper and Tofino compiler

To validate RMT resource mapper as a proxy for Tofino compiler, we conduct experiments to compare resource allocation results of RMT resource mapper and the Tofino compiler. We pick five different sketches (UnivMon, R-HHH, PCSA, HLL, and MRB). We vary one parameter of sketches while fixing other parameters and analyze the resource allocation results. We focus on five different resource types; pipeline stages, hash calls, SALU, SRAM, and TCAM.

Fig. 18–Fig. 22 illustrate the results. Note that all of the resource usages are normalized. We can see that for hash calls, SALU, SRAM, and TCAM usages are identical between RMT resource mapper and the Tofino compiler. For pipeline stages, results are the same for PCSA, HLL, and MRB. However, RMT resource mapper finds mapping which uses fewer pipeline stages than the Tofino compiler for UnivMon and R-HHH. RMT resource mapper minimizes stages while the Tofino compiler finds more sparse mapping (e.g., mapping a small number of tables per stage). We validate both of the mappings from RMT resource mapper and Tofino compiler are valid. We confirm with the vendor that the Tofino compiler uses complex heuristics and the cost function of power budget and compilation time, which are different from that of RMT resource mapper and can introduce the gap. Our extensions to the RMT resource mapper is available at [6].
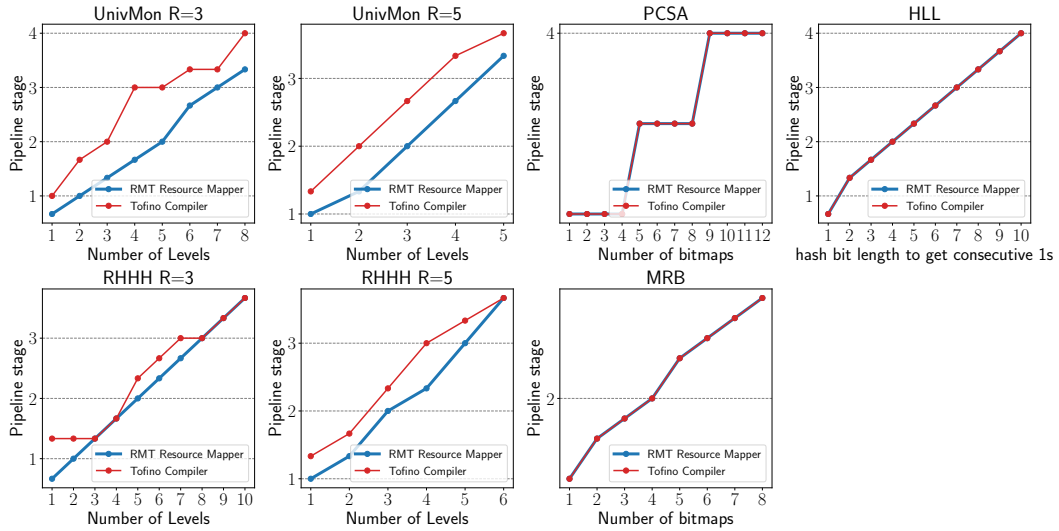
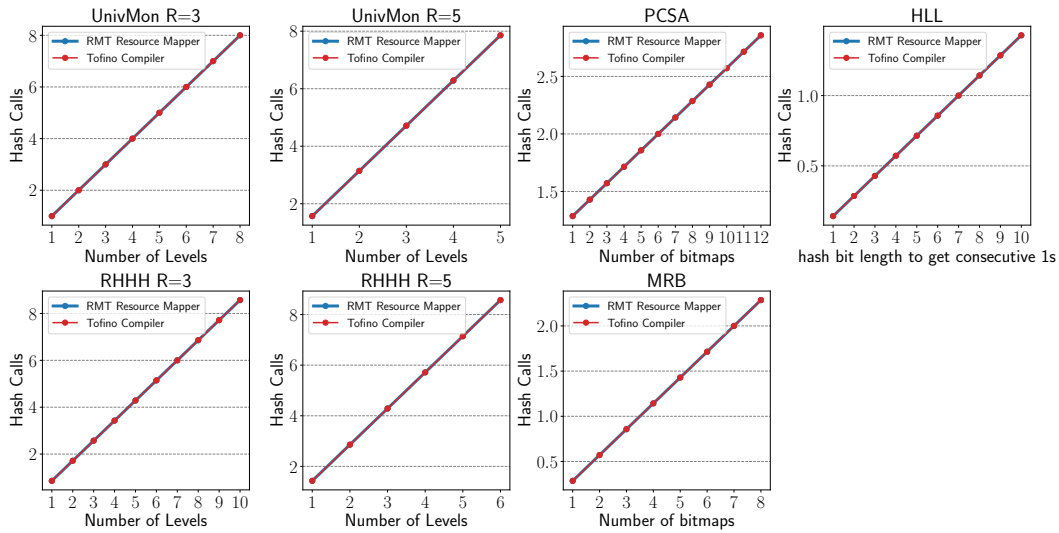**Figure 18: RMT resource mapper vs. Tofino compiler: pipeline stages**

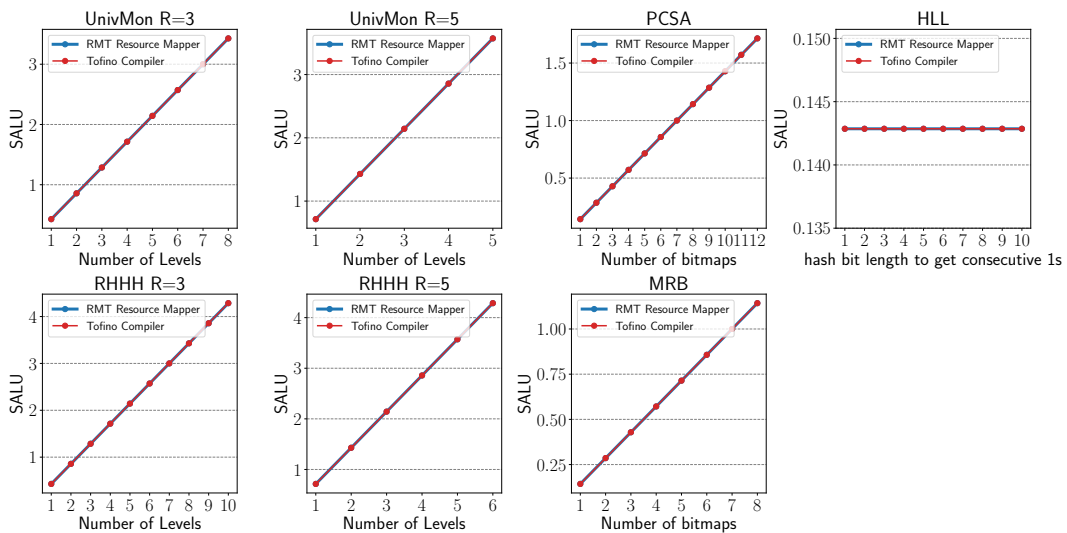**Figure 19: RMT resource mapper vs. Tofino compiler: Hash Call**

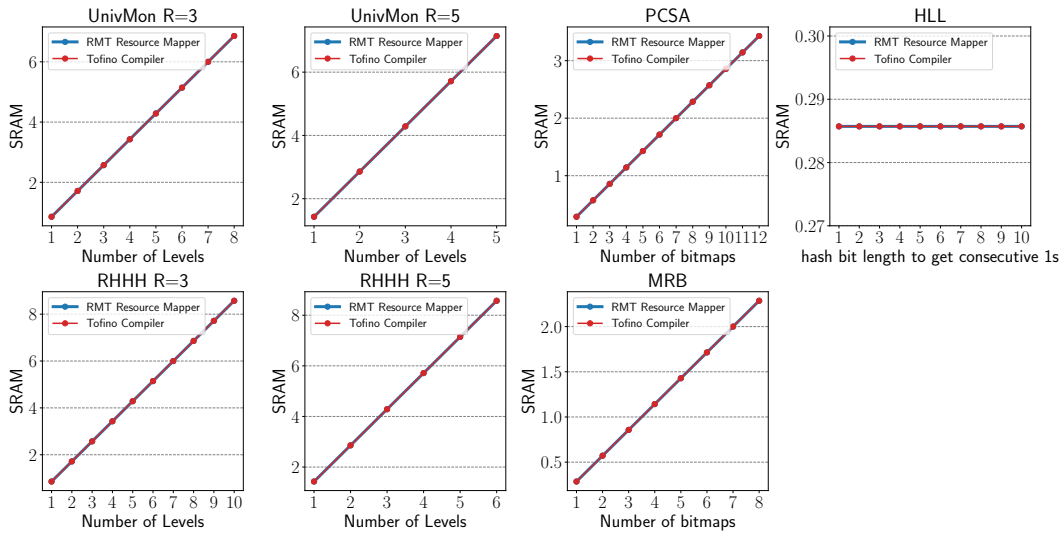**Figure 20: RMT resource mapper vs. Tofino compiler: SALU**

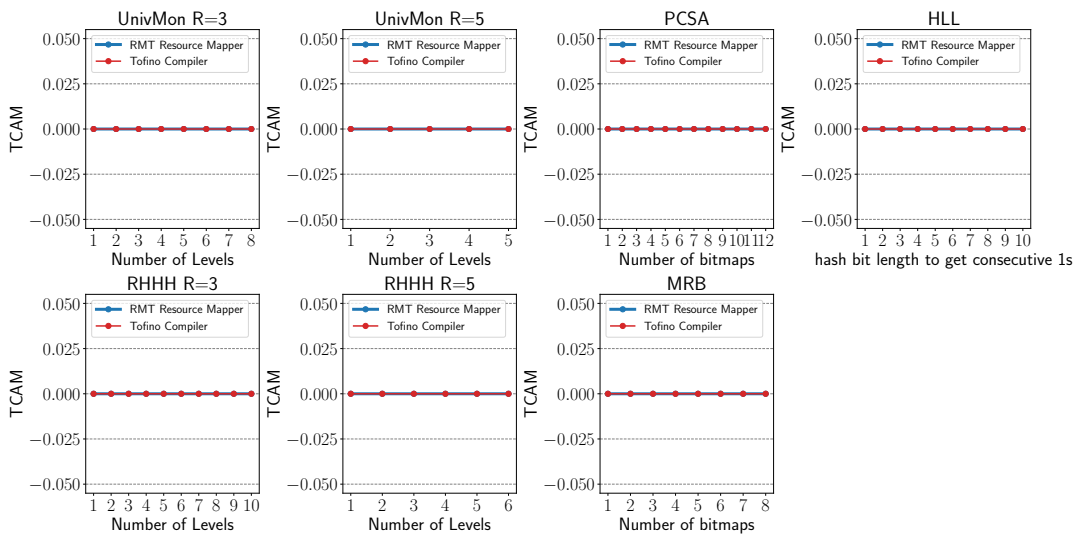**Figure 21: RMT resource mapper vs. Tofino compiler: SRAM**

Figure 22: RMT resource mapper vs. Tofino compiler: TCAM

An edge-queued datagram service for all datacenter traffic

Vladimir Olteanu *‡, *Haggai Eran*†#, *Dragos Dumitrescu*∗‡, *Adrian Popa*∗, *Cristi Baciu*∗,
Mark Silberstein†, *Georgios Nikolaidis*△, *Mark Handley*°∗, *Costin Raiciu*∗‡

∗ Correct Networks, † Technion, ° UCL, △ Intel, # NVIDIA, ‡ University Politehnica of Bucharest

Abstract

Modern datacenters support a wide range of protocols and in-network switch enhancements aimed at improving performance. Unfortunately, the resulting protocols often do not coexist gracefully because they inevitably interact via queuing in the network. In this paper we describe EQDS, a new datagram service for datacenters that moves almost all of the queuing out of the core network and into the sending host. This enables it to support multiple (conflicting) higher layer protocols, while only sending packets into the network according to any receiver-driven credit scheme. EQDS can transparently speed up legacy TCP and RDMA stacks, and enables transport protocol evolution, while benefiting from future switch enhancements without needing to modify higher layer stacks. We show through simulation and multiple implementations that EQDS can reduce FCT of legacy TCP by 2x, improve the NVMeOF-RDMA throughput by 30%, and safely run TCP alongside RDMA on the same network.

1 Introduction

Data center networks suffer from a range of unique problems that make it hard to effectively utilize the potential of the underlying high performance redundant multipath network topology. Notable issues include incast traffic patterns, flow collisions and transient congestion due to flow-level load balancing, interference between low-latency request/response traffic and bulk transfers, increasing requirements to offload work from the host CPU to avoid host stack bottlenecks, and the need to support special-purpose high performance protocols such as RDMA in the same network as legacy protocols.

These are all partially solved problems. There is a strong trend towards NIC offload, with datacenters deploying smart NICs and increasing ASIC support for specific transport protocols being offered by NIC vendors. However, moving transport state into NICs makes it harder for dissimilar protocols to coexist, and risks embodying the status quo in hardware.

At the same time, the research community has proposed a rich set of solutions such as phost[14], Homa[31], NDP [18], 1RMA [42] and Aeolus[20] which tackle incast and, to varying degrees, also address issues of load-balancing and low-latency request/response traffic. What these solutions share is a receiver-driven control loop that tightly manages inbound traffic, eliminating large in-network queues. Each of these, by

itself, would be a substantial improvement on the status quo, but datacenters cannot simply migrate to a single new transport protocol. Even if there were buy-in as to which transport protocol to adopt, there are far too many legacy applications and operating systems that would need to be re-written. How then can we take the best ideas from the research community and deploy them in production while supporting a plethora of legacy protocols ranging from vanilla TCP to RDMA?

One strawman solution would be to simply pick a low latency receiver-driven transport protocol and tunnel all datacenter traffic over it. The great advantage of such a control loop is that it performs admission control to the physical network, allowing very small switch buffers to be used while still providing low end-to-end loss. Is it possible to use this to provide a new datagram service that higher layer protocols such as TCP, DCTCP or RDMA run over?

The difficulty is that TCP, DCTCP, RDMA and other protocols each have their own expectations when it comes to sharing the underlying network. In particular, they use interactions between flows mediated via queues in the switches to drive their own control loops. We cannot just eliminate switch queues and expect everything to still work - rather we need to move the queuing from the switches back to the network edge, either in the host or NIC. The low-latency control loop can then clock packets from these edge queues into the network.

We have designed and implemented just such a layer called Edge-Queued Datagram Service (EQDS). Rather than a regular transport protocol, EQDS provides a datagram service to higher layers, implemented via dynamic tunnels. Its receiver-driven control loop is loosely based on NDP and 1RMA[42], but can be extended to utilize other in-network mechanisms where these are available.

Moving the interaction between flows out of the switch queues and back into EQDS edge queues provides many advantages. The receiver directly controls when enqueued packets from different senders enter the network, ensuring isolation even when higher layer protocols run different control loops. For example, TCP and RDMA will not normally coexist gracefully when sending to the same host, but EQDS can mediate, eliminating loss and allowing fair sharing.

Different EQDS queuing disciplines can be also used for different protocols, each providing appropriate feedback to the higher layer control loop; this both improves higher layer protocol performance, and it also allows dissimilar protocols sending from the same host to be protected from each other.
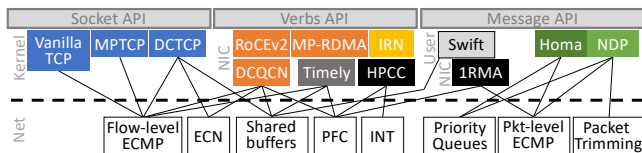

Figure 1: Fragmentation of datacenter networking

Adding new queuing disciplines to support innovative future transport mechanisms is also simplified as they only need to be deployed in the relevant sending hosts, not in switches.

Finally, EQDS uses packet spraying to balance load evenly in the network core, avoiding flow collisions, and increasing throughput. Legacy protocols such as TCP and RDMA do not normally cope well with reordering, so EQDS implements a reorder buffer in the receiver. With a conventional network such a reorder buffer might deliver the highest latency seen across all paths, but with good load balancing and short switch queues we find that EQDS reorder buffer latency is minimal.

In this paper we detail the design and implementation of EQDS and its on-demand zero-RTT tunnel protocol, and evaluate it running both natively in Linux hosts and offloaded to two brands of smart NIC. We show that the EQDS control loop operating on very short timescales does not adversely affect higher layer control loops such as TCP's Cubic or RDMA using DCQCN. Rather, it allows diverse higher-layer control loops to co-exist gracefully, protects latency-sensitive applications from queuing delays caused by bulk transfers, while increasing throughput by eliminating flow collisions.

2 Motivation

IP has been the narrow waist[7] of the Internet stack since from the early days of the Internet, providing basic end-to-end service. In reality, the narrow waist is not just IP: a functioning Internet also assumes some form of TCP-compatible congestion control and sufficient in-network queuing for it to do its job, though this lacks a clear layer in the stack.

This lack of abstraction has particularly hurt datacenter networking. Here, a plethora of work has pushed optimizations across boundaries, including to the host stack, switches or both. As a result of all these enhancements, what has emerged are multiple parallel stacks, each assuming a slightly different "basic" datagram service, that must be isolated from each other in the network to avoid them fighting (see Fig. 1). Further, optimizations for one stack often hurt the performance of others in the same network.

Many have improved on TCP congestion control[1, 28, 44], reducing vanilla TCP's need for large switch buffers. These TCP's are still built upon basic datagram service though, and probe network capacity to sense congestion. In so doing they interact with each other via queues, increasing latency.

Even datagram service itself has been tweaked, with many enhancements aimed at improving service for certain traffic classes, as in Fig. 1. For example, RoCEv2 can use PFC to provide lossless service as assumed by RDMA; this brings its own set of unique feature interaction problems [16, 29].

Protocols like TCP and RDMA also assume largely in-order delivery from the underlying datagram service. In datacenters, in-order delivery is provided by flow-level ECMP, though this wastes capacity in Clos topologies. To better use multipath networks, variants of these protocols have been proposed [38, 26, 29] but rarely deployed. Another source of performance problems as network speeds have increased has been the end-host stack implementation itself. Even TCP resorts to segmentation and checksum offloading, but application writers often use kernel bypass mechanisms such as DPDK or even offload all the work to the NIC using RDMA, and in so doing impose unique dynamic load on the network.

The web of dependencies between higher layer protocols and in-network enhancements makes deploying new protocols increasingly difficult. The root cause of the problem is that basic datagram service forces diverse higher layer protocols to interact via queues in the network. We argue that in-network queuing, beyond the minimum needed to smooth fan-in, is antithetical to building a high performance low-latency general-purpose datacenter network.

We propose a novel Edge-Queued Datagram Service as the new narrow waist for the datacenter networking stack. To transport stacks above, EQDS offers a what looks like a conventional datagram service via virtual interface queues in the host that buffer traffic and provide appropriate congestion feedback signals. EQDS then sends this traffic when possible, utilizing diverse in-network mechanisms to maximize utilization and minimize in-network queuing latency. EQDS shares the network at the hosts, allowing conflicting transports to run side-by-side on the same network.

3 Concept

We introduce the EQDS concept by means of example. Consider Figure 2a: two TCP senders send to a receiver across a conventional network where the bottleneck is at the final hop as is common in datacenters [5]. TCP needs to build a queue to sense congestion and back off, forcing any other flow sharing the queue to behave similarly to share the link reasonably. RDMA or other transports (see Figure 1) that do not will cause problems and need to be isolated from TCP.

In contrast, the EQDS concept is shown in Figure 2b. The underlying latency across modern datacenter networks is so low that protocols like NDP, Homa and Aeolus can use credit mechanisms whereby the receiver clocks packets from the sender as required, ensuring a standing queue never builds in the network. Protocols like TCP still need a queue to drive their control loop, but EQDS moves this queue to the sending hosts, where it is under the receiver's control. If many TCP senders create an incast, the default behavior they observe is almost identical to what would happen if the last hop switch runs fair queuing with a large amount of buffering.

As the queuing has been removed from the network itself, EQDS can run different queuing disciplines in the sending
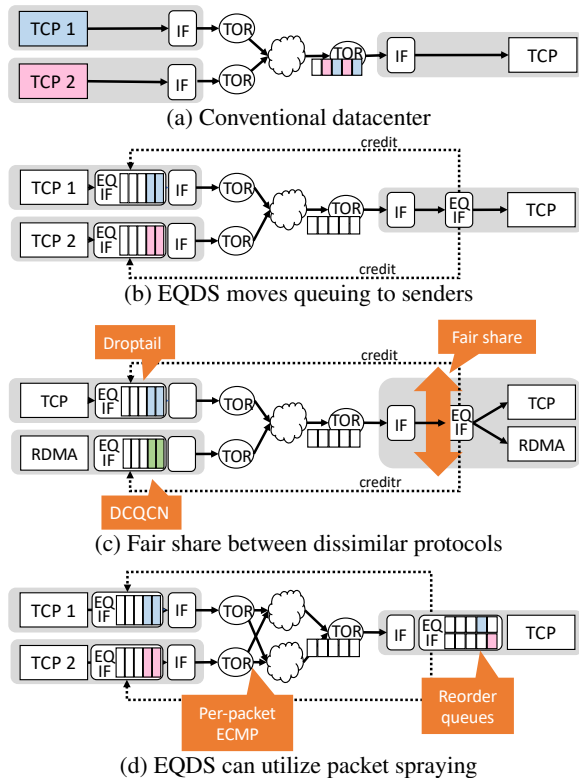
(a) Conventional datacenter

(b) EQDS moves queuing to senders

(c) Fair share between dissimilar protocols

(d) EQDS can utilize packet spraying

Figure 2: Overall EQDS concept

EQDS Virtual Interface (EQIF), as appropriate to the traffic being carried. Figure 2c shows a legacy TCP sender coexisting with an RDMA sender, with the bottleneck link being shared fairly, or with a proportional share, if that is deemed appropriate by the network or host administration policy.

Finally, Figure 2d shows EQDS using per-packet ECMP (aka packet spraying) in the underlying network to minimize latency and increase network capacity. Few current transport protocols cope well with the level of reordering this usually causes, but as EQDS keeps network queues very small, the reordering is easily managed by a short reorder queue in the receiving EQIF, so it is hidden from the higher level protocol.

In summary, the key EQDS concepts are: (1) move queuing out of the network leaving just the bare minimum required; (2) queue traffic in the sending host; release it when the receiver requests it; (3) run appropriate queue disciplines for different classes of application as they require; and (4) use per-packet ECMP to load-balance evenly so as to minimize latency but, by default, hide it from higher layer protocols. The EQIF virtual interfaces then become the control points for the network, enforcing sharing policies.

4   Design

To implement EQDS, we need four main components:

- One or more EQIFs on the sending host, which implement queuing disciplines to support higher level protocols;

- One EQIF on the receiving host, which implements a short reorder queue for best-effort in-order delivery service to protocols that are intolerant to reordering;

- A mechanism to encapsulate packets reliably across the network from sending EQIF to receiving EQIF;

- An edge-to-edge control loop to clock packets from sending EQIF to receiving EQIF.

With these in place diverse higher-layer protocols are carried over EQDS, which hides lower-layer in-network mechanisms. The edge-to-edge control loop may differ in different datacenter environments or even within one datacenter, depending on switch capabilities. In effect, EQDS has become the new narrow waist of the datacenter protocol stack.

In the virtualized protocol stack, EQDS operates at the same layer as VXLAN, encapsulating higher-layer traffic going to EQDS-capable destinations. To provide datagram service, EQDS needs to provide on-demand tunneling from EQIF to EQIF without prior setup, without spending an RTT performing a handshake to establish control state, and with the expectation that packets sent in the first RTT will be reordered by per-packet ECMP. This demands a novel tunnel protocol (§5). EQDS tunnels are unidirectional; two are setup, one in each direction, if user traffic is bidirectional.

EQIF per-destination tunnel state is established on packet receipt at the sender, and established at the receiver using a zero-RTT protocol. This state can be unilaterally discarded when idle at any time by either side to reduce memory usage and will simply be reestablished as needed.

4.1   EQDS control loop

Critical to EQDS performance and minimizing in-network queuing is the edge-to-edge control loop. EQDS allows different control-loop mechanisms to be used depending on the underlying network capabilities.

For a fully provisioned network our preferred in-network mechanism is packet trimming, which allows EQDS to use an NDP-derived control loop. This allows a burst of packets to be sent in the first RTT before credit-based control from the receiver takes over for subsequent RTTs. The sending EQIF keeps packets until they have been acknowledged by the receiving EQIF, or retransmits them on receipt of a NACK. In this manner, EQDS provides a highly reliable service, but it does not guarantee no packet loss whatsoever. A full reliability guarantee would prevent EQDS managing its own state effectively, risk resource starvation attacks, and would be pointless as full reliability requires end-to-end acknowledgment whereas EQDS may be implemented in the NIC so cannot protect data all the way to the receiving process.

Where packet trimming is unavailable, EQDS uses a 1RMA[42]-derived mechanism where the sending EQIF requests credit from the receiver, or it can use a Homa/Aeolus-derived mechanism where the first RTT of data is sent using
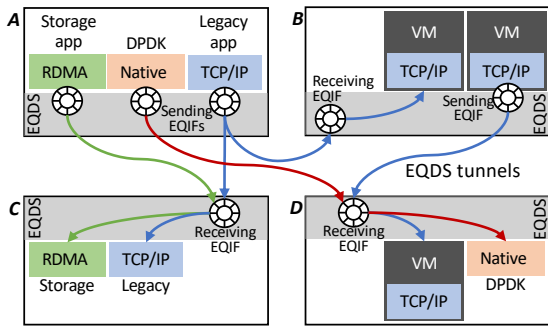
Figure 3: Multiple stacks run on the same substrate.

Figure 4: Tunnel Headers for Data / Control Packets

low-priority service. In an underprovisioned network, core network congestion is also possible, though it should be rare if per-packet ECMP is used. In such cases, in-band network telemetry (INT) might by used to implement an HPCC-style control loop. Our current implementation supports both NDP-style pro-active transmission and request-to-send. We expect future networks to innovate further in this area.

EQIFs. EQDS allows multiple protocol stacks to run on the same network substrate, as shown in Figure 3. The DPDK-based stack on host *A* is sending to its peer at *D* while a virtual machine at *B* also uses TCP to send to the VM at *D*. Without EQDS, the DPDK stack will saturate the link to *D*, starving TCP. This could be prevented using fair queuing in *D*'s ToR switch, but EQDS achieves the same effect without needing to configure switches and with finer grain control.

At *D*, a single *receiving EQIF* receives the flows from *A* and *B*. It maintains state (reorder queues, sequence numbers, etc) for incoming EQDS tunnels, allowing it to effectively manage all incoming traffic. By default, *D*'s receiving EQIF will send equal credits to *A* and *B*, ensuring a fair share and no overload. Proportional sharing or strict priority can also be achieved: the inbound sharing policy can be configured as needed - both by the network administrator and, if the VM and native stacks are run by the same user, by them too.

Three different stacks are in use at *A*: there are RDMA and TCP flows to *C*, a TCP flow to a VM at *B* and the DPDK native flow to *D*. This results in three *sending EQIF* virtual interfaces at *A* for the three different edge queue disciplines. The two TCP flows share the same sending EQIF which runs a TCP-compatible queue discipline. All three EQIFs at *A* cooperate using deficit round robin, so if the total traffic saturates *A*'s outgoing link, they will share it fairly (or unfairly, if that is the configured policy).

To summarize: an EQIF is a virtual interface that implements a specific queue discipline or feedback mechanism to higher-layer protocols. One sending EQIF contains multiple queues, each feeding an EQDS tunnel to a single host. Multiple transport flows from multiple VMs can share one EQIF so long as the protocols used can coexist in the same queue.
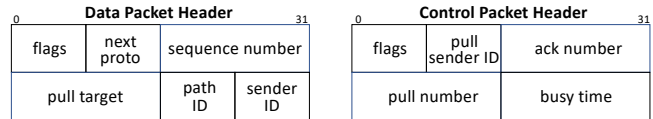
5 Tunnel protocol

To carry data from sending EQIF to receiving EQIF we need a new tunnel protocol. The primary requirements are:

- A receiving EQIF should clock packets from its set of sending EQIFs so as not to cause in-network queues to build. At the least, this means it will send credit at a rate that does not exceed the receiver's access link speed.

- A receiving EQIF can choose how to distribute credits to senders, with the default being to implement a fair share.

- The tunnel should expect per-packet ECMP service and be robust to reordering this causes. Where possible, the sending EQIF will determine the path taken by each packet.

- The tunnel should provide best-effort reliable and in-order delivery from the receiving EQIF to higher layer protocols. Losses and reordering should be rare enough to minimally impact the performance of higher layer protocols.

- The tunnel should support unreliable, out-of-order delivery to higher layer protocols that prefer minimal latency.

- The tunnel should come up on demand with no pre-data handshake required, and be discardable at any time. If the endpoints end up out of sync, it should self-synchronize.

- The tunnel should be able to take advantage of a range of underlying network enhancements without higher layer protocols needing to be aware of them.

- Both sending and receiving EQIFs should be able to impose policies for sharing, configured by the network operator and by the users (to the extent user policy does not conflict with operator policy).

- EQIFs must be capable of being implemented in fast NIC hardware with bounded memory resources.

These requirements necessitate an unusual tunnel protocol; it has many aspects of a transport protocol, but is soft-state, being established, dropped and reestablished based on packet arrivals, yet it provides fine-grain closed-loop control.

At a basic design level, EQDS is a tunneling protocol with an NDP-derived control loop, that runs on top of UDP. It can carry multiple types of traffic, including IP and VXLAN. Its control fields (shown in figure 4) were meant to complement VXLAN and there is no overlap in functionality between the two; indeed, EQDS could be implemented as a stateful extension to VXLAN encapsulation if required [1].

Data Clocking. As with NDP, the receiving EQIF sends PULL packets containing credit to the sending EQIF; the

---

[1]For testing, we encapsulated plain IP traffic, rather than VXLAN.

sending EQIF then sends data packets matching that credit from the corresponding tunnel queue in response. The receiving EQIF paces the sending of credit so that after the first RTT the aggregate arrival rate matches the incoming link speed.

To summarize NDP as described in [18]: when a sender starts, one RTT of data is sent without waiting for credit; after that, the sender waits for PULL packets from the receiver. This means that there can be an incast in the first RTT where loss occurs. NDP copes with this using *packet trimming* - the payload is removed from packets that would overflow the queue, and just the header is forwarded to the receiver. This makes the network lossless for metadata, though not for data, and informs the receiver of demand. The receiver can then request retransmission of trimmed data and transmission of new data using PULL packets. In a large incast it may take some time to send a PULL to a sender, so the receiver ACKs or NACKs each data packet so that the sender knows which packet buffers to free and which to add to its retransmit queue.

Where the underlying network supports trimming, an EQDS tunnel uses the NDP mechanism as described above for the first RTT, as it minimizes latency. EQDS supplements this with a request-to-send mechanism, where the sender directly requests credit from the receiver. This is used when trimming is unavailable and for intermittent bursty flows.

Conceptually, each EQDS tunnel maintains a constant bandwidth-delay product (BDP) of credit which is passed between sender and receiver. This credit either starts at the sender (NDP-like) or at the receiver (RTS). Credit flows from sender to receiver with data packets and from receiver to sender with PULL packets. EQDS differs from NDP in how it keeps this window constant in the presence of control packet loss, as NDP failed to do so in corner cases.

EQDS credit is expressed in bytes. To send a packet of size b bytes, the sending EQIF must possess b bytes of credit. PULL packets contain a *pull number* which starts at zero and increments for each PULL sent to a source. When the highest pull number seen by the sending EQIF increases by n, this grants n MTUs of credit.

When a sending EQIF sends data, a *pull target* field in the header indicates to the receiver how much credit is desired beyond the current pull number. This is capped at one bandwidth-delay product (BDP) - typically 10 to 30 packets.

The receiving EQIF maintains an *active sender list* (ASL). An active sender is an incoming EQDS tunnel that has outstanding data to send. Every MTU-time the receiving EQIF will send one MTU of credit to the sender at the head of the ASL. If this causes the pull number to reach that sender's pull target, this credit will satisfy all the known demand from that sender. The sender will then be removed from the ASL and placed in an inactive senders set. If the pull number does not reach the pull target, the credit sent will not yet be sufficient, so the sender is re-inserted at the tail of the ASL. In this way all active senders get a fair share of capacity and credit is not sent to sending EQIFs that have no queued data.

The ASL is similar in concept to the NDP pull queue, but unlike NDP it ensures that a one-BDP credit window invariant always holds. Conceptually, the sum of credit stored at the sender, packets in flight, pulls in flight carrying credit, and credit implicitly stored in the ASL entry at the receiver is a constant so long as sufficient demand remains.

Implementing the ASL as a FIFO ensures incoming traffic is split fairly by default. To implement other sharing policies, a PIFO queue[43] can be used in place of a FIFO, allowing a wide range of policies to be implemented.

To avoid sources going idle and then immediately bursting again, the receiving EQIF tells senders the minimum time its access link will be saturated using the busy time field in control packets (the pull targets inform the receiver of the total queue size at every sender). Even in trimming networks, if a bursty sender restarts within this busy time, it always uses RTS before sending, as bursting would cause unnecessary trims; senders can burst after the busy time elapses.

Tunnel setup and teardown. A sending EQIF creates tunnel state when packets for a new destination arrive. It picks a sequence number with certain constraints (see Appendix A for details) and starts encapsulating packets without waiting for a handshake to complete, setting the SYN flag in all packets until it receives a matching SYN+ACK packet in response. This informs the receiver of the new tunnel and is robust to reordering caused by per-packet ECMP.

Either side can unilaterally drop tunnel state. As an optimization, each will inform the other when it does so, but such a teardown does not need to be signalled reliably, and neither end keeps time-wait state. Later, if new packets arrive at the sending EQIF, a new tunnel will be established. If a receiver tears down a tunnel from a sending EQIF that has queued packets, a new tunnel is immediately set up.

This simplicity allows the simple EQDS state machine in Appendix A, it allows EQDS to be self-synchronizing if the two endpoints end up in different states, and it minimizes state requirements - something that is important for EQDS implementation in hardware. We can get away with such a lightweight protocol because the EQDS service model only guarantees a best-effort attempt to avoid loss, duplication, or reordering. A conventional transport protocol like TCP needs to provide firmer guarantees to the application.

Reorder Queue. Per-packet ECMP greatly improves load balancing, reducing in-network queuing and latency, but may cause reordering. Trimming also causes reordering while awaiting retransmission. To avoid performance problems with higher-layer protocols, EQDS maintains a per-tunnel reorder queue in the receiving EQIF. With minimal in-network queuing, the delay difference between paths is small, so this queue does not grow much and is bounded by a BDP.

Oversubscribed networks. When the network core is oversubscribed and becomes a bottleneck, aggressive receiver-driven transports can result in high trim rates or in high la-
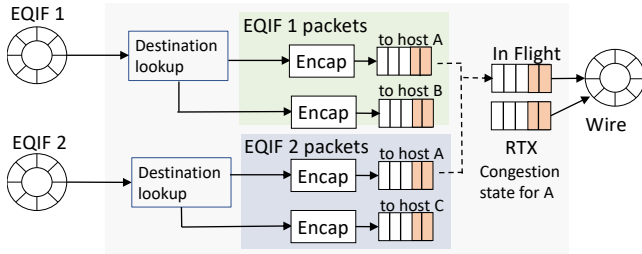
Figure 5: Transmit datapath for sending EQIFs

Figure 6: RDMA EQIF implementation.

tency or loss when RTS is used. In such cases, EQDS will need to be enhanced to take into account other congestion signals such as ECN, latency or even in-band network telemetry[32] *in addition to* receiver pacing. As the receiver-driven transport handles the high-dynamic-range incast case, such congestion management only needs a relatively limited dynamic range and can be implemented by either the receiver reducing its pull rate or by then sender reducing its pull target. Developing such mechanisms is future work.

Incremental deployment. EQDS only encapsulates traffic to configured internal address ranges, so external and legacy traffic will also be present in a datacenter. How will they coexist? EQDS's packet spraying diffuses the effects of a flow across many core links, greatly reducing its impact on any legacy single-path flow. In our testbed when trimming is enabled, we use two priority classes to separate EQDS and non-EQDS traffic and ensure low latency for EQDS via small buffers even in the presence of legacy "elephant" flows.

Strict prioritization is probably undesirable as load levels rise, but weighted fair queuing between EQDS and non-EQDS traffic classes can maintain low latency for EQDS flows in the core, so long as the sprayed load-balanced EQDS aggregate does not exceed its allocated share. On ToR uplinks where EQDS traffic is less diffused, legacy "elephant" flows may impact some EQDS paths more than others. EQDS offers accurate per path latency and loss statistics that can be used to perform load-adaptive routing between paths, avoiding transient bottlenecks. Implementing these is future work.

6 Sending EQIF Specialization

Different types of traffic have different expectations of the underlying datacenter network. While a single EQDS tunnel protocol clocks all traffic from sending EQIF to receiving EQIF, higher-level protocols with differing network expectations are supported by different specialized sending EQIFs.

Fig. 5 shows how sending EQIF behaves as a virtual interface. Whenever a higher-layer protocol sends packets via that interface, EQDS encapsulates and enqueues them, pending sufficient credit being available. The sending EQIF maintains one queue per tunnel, allocated on demand if one does not already exist. When a packet is sent, it is moved from the send to the in-flight queue, but not freed until it is ACKed. If is it
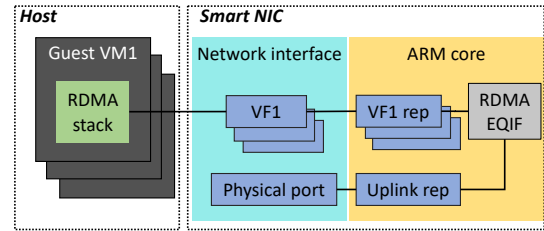
NACKed or a retransmit timer expires, it is moved to the tunnel's retransmit queue. When credit is available, retransmitted packets are sent first, then new ones.

Each sending EQIF provides a specific type of service. We currently support three service classes: TCP-compatible, RDMA, and native. They differ in their queue discipline and in how feedback is presented to end-to-end traffic.

When multiple sending EQIFs are in use at the same host, as in Figure 5, credit avoids overloading the receiver's downlink, but credit from multiple receivers can exceed the uplink speed. When this happens, deficit round robin is used to share the physical interface fairly between the various EQIF queues. Queue priorities can also be configured if desired.

Multiple host stacks or virtual machines sending the same traffic class can share the same EQIF. For example, QUIC and TCP may use the same TCP-compatible EQIF, whereas RDMA would use a different EQIF. None of these legacy stacks need be aware they are running over EQDS.

TCP-compatible EQIF Class. Our TCP EQIF implements a simple drop-tail queue for non-ECN traffic and a RED queue for ECN-capable traffic. The goal of the queue is to absorb traffic when TCP is sending faster than the receiver wants. When the queue fills, a packet will be dropped. The queue needs to be large enough that TCP's congestion control can operate and saturate the receiver's link - typically this will be upwards of 30 packets.

The worst case for TCP is when many flows incast to the same receiver. With a default ten-packet initial window, packets will be queued in the EQIF queue until the receiver sends credit, which may take some time. TCP's 250 ms minimum RTO time helps here - even large incasts can usually complete within 250 ms. If packets are queued longer than this, an RTO may occur, but our experience is that even this has little impact on performance; TCP detects a spurious timeout via the Eiffel algorithm[37, 36], corrects its congestion window, and updates the RTO to prevent further timeouts.

We find that vanilla TCP running over EQDS almost always outperforms TCP running natively, even when not competing with incompatible flows. Much of this win comes from EQDS's use of per-packet ECMP.

RDMA EQIF Class. RDMA requires a separate EQIF to avoid fighting with other traffic, to avoid loss seriously impacting RDMA performance, and to provide appropriate flow control feedback to the RDMA implementation.

RDMA is typically implemented in the NIC. Ideally an RDMA EQIF implementation would be coupled with the hardware transport implementation to directly flow-control RDMA traffic. We currently deploy our prototype RDMA EQIF in a smart NIC, as shown in Fig. 6. We use port representors [8] to interpose on RDMA-enabled devices the SmartNIC exposes to the host and its virtual machines.

Our EQIF does not modify the NIC's RDMA implementation, but it does need to tell the sender to slow down when the TX queue to a destination grows. Depending on the SmartNIC model, we use different techniques to control the sending rate of the RDMA engine. For the Stingray, we issue PFC PAUSE packets to slow down the sender; this works well, but it has the side effect of slowing all traffic coming out of the RDMA engine, not just the one to the backlogged destination. The BlueField 2 supports DCQCN, so our implementation uses this to control RDMA. We could use ECN to signal DCQCN flows to slow down, but we prefer to send congestion notification packets (CNPs) directly from the sending EQIF to the RDMA sender. This reduces the length of the DCQCN control loop and allows one-time tuning of the DCQCN parameters to this constant delay. Unfortunately, RDMA RC packets lack the source QP number in their headers, which is needed for sending back a response, so we develop a connection tracking module [10] for RDMA CM, enabling CNP generation.

Our current smart NIC implementation moves packets from the host to the ARM cores and then to the wire; with bidirectional traffic the SmartNIC's interconnect can become a bottleneck. It should be possible to only move the packet headers to the ARM cores, but our implementation does not support this yet. To avoid our results being affected we configure our RDMA testbed to a lower rate (10 Gbps). A NIC designed for EQDS would not add this additional latency. Despite this, our implementation increases RDMA performance in many cases while allowing coexistence with other protocols.

The Native EQIF is the preferred option for performance-oriented, EQDS-aware transports. It uses a shared memory area to store packets with lockless descriptor rings used to move packets to and from the EQIF via a zero-copy API. This EQIF offers additional low level information to host transports including the size of the TX buffer to the destination, the size of the destination's pull-queue, per-packet and per-path network RTTs and delivery notifications.

Latency information provided by the Native EQIF is similar to that provided by 1RMA, so for cases where core congestion is common, delay-based congestion controllers such as Swift [24] or Timely [28] can be implemented on top.

We have implemented *eqdsperf*, a performance testing tool over the Native EQIF, as well as a lightweight UDP stack that is optimized to run over EQDS. Applications using the UDP socket API can simply be linked to our stack, either statically at compile time, or dynamically using LD _PRELOAD.

7   Implementation

We implemented two versions of EQDS, one using DPDK and one as a Linux kernel module. The goal is to add minimal overhead, but inevitably there are tradeoffs to be made.

Our DPDK implementation was built with performance in mind and uses two CPU cores, regardless of load. These can be host CPU cores, but it is preferable to use the ARM cores on a smart NIC. The main thread takes turns reading packet batches from the host-facing and network-facing NICs. Once read, packets are processed to completion. At the sender, this includes NACKs and PULLs triggering the (re)transmission of queued packets, and at the receiver trimmed packets elicit NACKs while packets that fill a hole at the head of the reorder queue trigger the release of waiting packets. The main thread also checks for timer expiration, largely eliminating the need for synchronization. A second thread is used to pace PULLs and to send deferred ACKs, providing accurate PULL pacing at the expense of burning a second core. When the NIC supports fine-grained pacing (e.g. Intel Columbiaville), EQDS can offload pull-pacing to the NIC, reducing CPU usage.

Our kernel implementation is aimed at being cheap and easy to deploy. As is usual on Linux, outgoing packets are processed in the context of the sending process. They are captured by a hook after routing has taken place. If there is enough credit, processing continues until they are handed to the NIC's driver. Inbound, EQDS acts like a UDP service; all incoming packets, both data and control, are processed in a soft IRQ and fed to a kernel UDP socket. Data packets that can be forwarded straight away are re-injected into the IP stack after decapsulation. If control or data packets have to be sent back, a Layer 3 socket is used.

High Resolution Timers are used for PULL pacing with their handlers executed in a soft IRQ. The downside of using kernel timers is that their timing is at the mercy of the Linux scheduler. This adds jitter to the PULL pacing. To mitigate such jitter, senders must use higher window values than usual.

Offloads are key to achieving high TCP performance with Linux, so our implementation leverages both TSO and GRO. With TSO, TCP will send large segments, EQDS will encapsulate them, then an EQDS-unaware NIC will split the encapsulated packet, copying the extra headers verbatim in front of the inner TCP/IP headers. The problem is that all the split packets will have the same EQDS sequence number. Fortunately, due to a peculiarity in how TSO is performed, there is a workaround. A NIC increments the IP ID of every segment following the first. We send all EQDS packets with an IP ID of zero, and leave gaps in the EQDS sequence space. The receiver then adds the received IP ID to the received sequence number to obtain the full sequence number. Future EQDS-aware NICs would remove the need for this workaround. Inevitably, using TSO causes a burst of unpaced packets to be sent which, as with timer jitter, requires a small increase in the window used by EQDS.
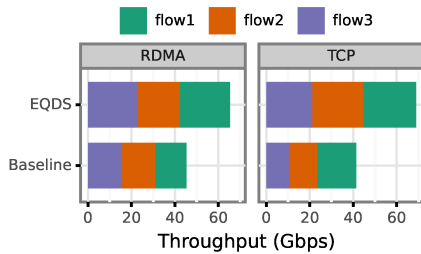
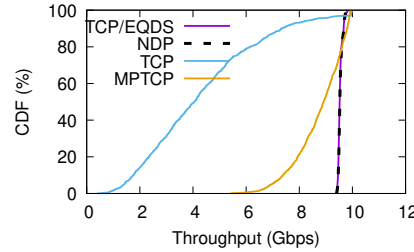Figure 7: Permutation throughput in the T2 testbed (BlueField-2 hosts).

Figure 8: Simulation: permutation throughput in a 1024-node Fat Tree.
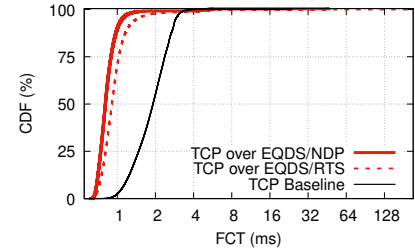

Figure 9: FCT: 1MB flows from random servers, closed loop (T1 testbed)

8 Evaluation

Datacenter networks have been shown to suffer from a number of pathologies including reduced throughput due to flow collisions, high loss or delay due to incast, inflated latency for RPC-style traffic and a dependency on compatible congestion control mechanisms for network sharing. EQDS's main goal is to enable the deployment of known solutions to all these problems in actual networks, and to make the resulting performance available to unmodified host stacks.

The first part of our evaluation examines whether EQDS helps mitigate these known pathologies and can boost performance for regular datacenter applications. To ensure the results are not specific to one host stack or deployment model and do not depend on network support, we experiment with unmodified TCP/IP (Linux) and RDMA NIC stacks (Blue-Field 2, CX4), run EQDS both on the host and on two smart NICs, and use both legacy and trimming-enabled networks.

We find that EQDS helps boost throughput of unmodified TCP/IP and RDMA stacks by up to 30-40%, mitigating the effects of collisions in a permutation traffic matrix (§8.1) and for NVMe over Fabric traffic. It achieves near-perfect incast behaviour with very small in-network queues both with trimming (our testbed and simulation) and RTS (on Amazon EC2), halving the latency and doubling the throughput of a micro-service based social network application benchmark on busy networks, as well as speeding up memcached by 10-30x (§8.2). Finally, we show that EQDS helps conflicting upper-layer congestion controllers nicely share the underlying network without any in-network support (§8.3).

In the second part we seek to understand whether EQDS introduces problems of its own. The biggest concerns are host overheads such as EQDS software's memory and CPU costs, its ability to handle high link speeds and dependence on specific hardware for performance.

In our evaluation we used two small-scale testbeds: T1 is a testbed we used for TCP/IP tests (10 servers) and T2 for mixed RDMA and TCP/IP tests (6 servers). Both testbeds used leaf-spine topologies and support trimming, but can also be configured with drop-tail/ECN; a detailed testbed description is provided in Appendix B. To test behaviour at scale and behaviour over legacy networks we use a large scale deployment in Amazon EC2 as well as simulation.

8.1 Improving throughput

TCP and RDMA require in-order packet delivery to function well, so datacenters switches hash the packet 5-tuple to select one of many equal-length paths to the destination. However, when the number of flows is small and the flows are high bandwidth, such placement can randomly place multiple flows on the same link causing congestion. Prior work has shown that flow collisions degrade performance in folded Clos topologies [11, 38] by up to 60% in the worst-case where a permutation traffic matrix is used, where each host sends to and receives from one other host. EQDS's per packet multipath should avoid such performance loss, at the cost of performing reordering in the receiving EQIF. We have run permutation experiments for RDMA in testbed T2 and for TCP in testbeds T1 and T2 as well as in simulation at larger scale.

Figure 7 shows that EQDS successfully spreads single flow TCP and RDMA traffic over all the available paths without causing reordering problems. TCP and RDMA flows running over EQDS achieve 22Gbps on average (maximum 24Gbps) in a permutation traffic matrix, compared to an average of 12-14Gbps without EQDS.

Our simulation results in Figure 8 explore behaviour at larger scale (FatTree with k=16, 1024 servers, 10Gbps NICs). Flow collisions hurt TCP badly, yielding only 40% of the network capacity. Multipath TCP[38], a variant of TCP that spreads traffic over multiple subflows (8 in our experiment) fares better with mean utilization close to 90%. NDP's packet spraying enables it to achieve near-optimal throughput. Finally, TCP over EQDS benefits from packet spraying but avoids the costs of reordering which is handled by EQDS, achieving similar performance to NDP.

Permutation traffic matrices highlight worst-case behaviour, with infinite flows and the smallest number of flows possible. Do collisions actually matter for other traffic matrices, for shorter flows, and for real applications? We examine this next.

1MB flows, random traffic matrix. On our testbed we run a workload where each server downloads a 1MB object from another randomly chosen server in a closed-loop, mimicking a storage workload over TCP. We measure flow completion times and plot them in Figure 9. EQDS lowers median and 95th percentile completion times by 2.4x and 2x respectively.

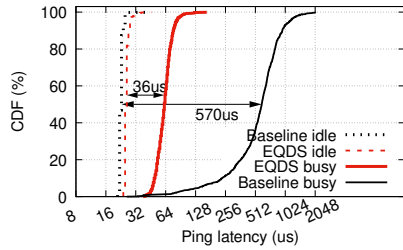We also ran the same experiment with EQDS using request-

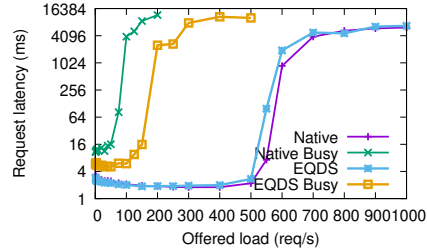Figure 10: Ping latency: target is idle or busy with 9 incoming TCPs (T1 testbed).

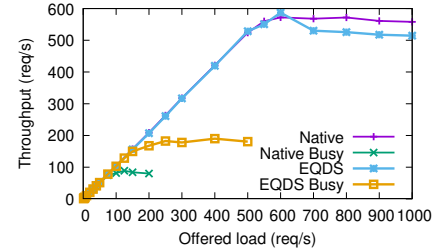Figure 11: DeathStarBench in the T1 testbed: request latency

Figure 12: DeathStarBench in the T1 testbed: throughput

to-send and without trimming support in the switches. The flow completion times (FCT) for this setup are, as expected, slightly larger than the NDP-based implementation due to the additional RTT needed at the beginning of transfers. Still, EQDS/RTS halves the completion time of TCP compared to baseline, both in the median and at 95%. At 99% all variants have similar FCTs, and beyond that EQDS can take longer due to missing optimizations in the code for short flows.

NVMeOF. We run a disaggregated storage service using a Storage Performance Development Kit (SPDK) NVMeOF-RDMA target with a null block device (i.e. no storage access) to stress the network subsystem. The NVMeOF-RDMA protocol is target-driven, with the server reading or writing to memory buffers in the client after coordinating the access via rendezvous. This involves both latency-sensitive operations for control and throughput-hungry operations for data.

We run the NVMeOF targets and clients on separate ToRs. Each client accesses the targets round-robin, using the SPDK `perf` utility to generate the workload. Figure 13 shows the throughput of random writes and reads of 64 KB blocks while varying the queue depth of the NVMeOF target. EQDS with its multipath support increases both peak read and write throughput and reduces standard deviation. Note that EQDS requires deeper NVMeOF queues to achieve higher throughput due to the added latency of our BlueField setup (§8.5).

8.2 Improving application latency

In deployed networks, there is a strict tradeoff between supporting many-to-one incast traffic gracefully, typically by provisioning large shared buffers in the network, and the latency of request-response applications such as micro-services or in-memory key-value storage (e.g. memcached). EQDS solves this trade-off by moving the buffering to the sending hosts, promising to achieve both low latency and good incast behaviour simultaneously.

Many to one traffic. To understand the baseline behaviour, we run large many to one workloads (850 iperf senders to the same receiver) in both simulation and on Amazon EC2 VMs.

We use htsim simulation to understand the behaviour of TCP NewReno when the destination link runs at 10Gbps, and we vary the size of bottleneck switch buffer. Figure 14 is
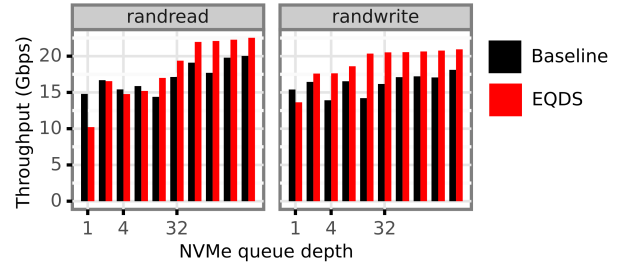
Figure 13: NVMeOF in the T2 testbed (BlueField-2).

a CDF of the flows' mean throughput. When the buffer is small (approx. 1 packet per flow), there is a large variance of throughput, with many flows starved. As we increase the buffer to around 100 packets for each flow, TCP can share capacity much more evenly, and there is no starvation. We also show the result when the switches implement fair-queuing, which further reduces variance and reduces buffer needs.

We run the same workload over EQDS, with the EQIF per-destination buffer set to 100 packets and a 15 packet buffer at the bottleneck. EQDS perfectly shares the bottleneck capacity, emulating a fair queue at the bottleneck link.

While our simulations show that TCP many-to-one requires significant buffering to work well without EQDS, what is the actual behaviour in production datacenters? We rented VMs on Amazon EC2 (m5.8xlarge for the receiver, and m5.xlarge for all others, 10Gbps link speeds) and deployed the Linux kernel version of EQDS, configured to run in RTS mode, as EC2 does not currently implement trimming.

We ran the same workload, with 850 hosts running iperf to one receiver and plot the results in Figure 15. Note the linear x axis for this plot: many to one traffic in EC2 works fairly well, similar to the simulated fair-queuing results. TCP running over EQDS in EC2 achieves almost perfect sharing and 4% better throughput than the baseline, matching our simulations.

To achieve such good sharing, it appears that EC2 uses large buffers. We measure these buffers by pinging the same destination from an idle VM before and during the incast. Figure 16 is a CDF of ping latency. EC2 ping latency increases 163x from $55\mu s$ to 9ms during the 10Gbps incast, indicating that a buffer of around 11MB exists in this network. In contrast, EQDS achieves similar many-to-one throughput with only a $21\mu s$ increase in ping latency.

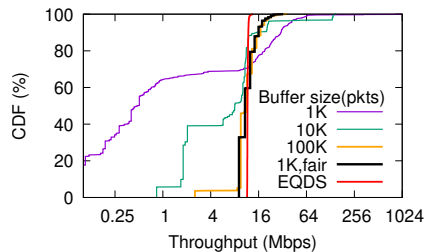Is this behaviour specific to EC2 or the RTS backend? We

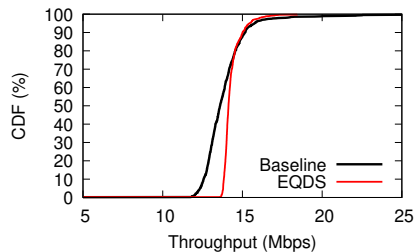Figure 14: Simulation: 850 to 1 traffic, varying size of bottleneck buffer.

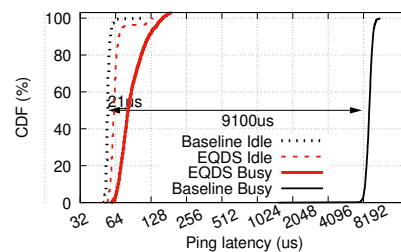Figure 15: Amazon EC2: 850 to 1 traffic, kernel EQDS, RTS mode

Figure 16: Ping latency (EC2): target is idle or busy with 850 incoming TCPs.

ran a similar experiment in our testbed using trimming instead of RTS. Figure 10 shows the results. When the destination is busy, EQDS with trimming results in a 36μs increase in ping RTT, similar to RTS on EC2. Without EQDS, the increase is around 500μs because our switch buffers for TCP are 1MB (less than EC2) and the link speed is 25Gbps. In summary, EC2 appears to be using large buffers to cope with incast. How does this affect latency sensitive traffic?

Memcached. We installed `memcached` on our EC2 destination VM and used `memcslap`, a benchmarking client, to measure server performance by issuing 1000 GET/PUT operations in a closed-loop manner. The mean request latency for an idle memcached server in EC2 is 700us. When running over EQDS, the same request takes around 900us due to the additional kernel processing EQDS does and the use of RTS.

When the destination is busy with 100 iperf clients sending to it, the mean request latency over EQDS increases 3x to 3ms, mostly due to sharing bandwidth with iperf, compared to a 140x increase to 100ms without EQDS due to large buffers. With 850 iperf clients sending, memcached over EQDS has a mean request latency of 23ms due to sharing bandwidth with 8.5 times more flows, compared to 400ms over native EC2. Overall, EQDS reduces memcached request latency for a busy EC2 server by 20 to 30x compared to the baseline.

Micro-service apps. We ran the latency-sensitive social network application of DeathStarBench [13] over kernel EQDS. We distributed the micro-service nodes to ensure that most requests are not local and must use the network, and used `wrk` to generate requests in a closed-loop manner.

We tested two scenarios: one where the network was idle, and one where high-throughput traffic going to the same hosts saturated the links filling the switch buffers. We note that the social network application does not generate much traffic – 100Mbps peak – but is latency sensitive so we expect to see an impact of longer network latencies in our "busy" scenario.

Figures 11 and 12 show the results. On an idle network social network requests take ~2ms to complete, with little difference between the baseline and EQDS. Our deployment can sustain a request load of about 500 requests per second after which DSB saturates the CPU. When the network is busy, however, EQDS reduces request latency by 50% and can sustain double the baseline request load.

8.3 Sharing the network

EQDS allows fine-grained host sharing policies without in-network support; we examine some interesting scenarios.

Non-responsive traffic competing with TCP. Consider the scenario in Figure 17 where a receiver in our testbed receives data from 4 TCP senders and then a UDP sender starts sending at line-rate (emulated with iperf3). As the UDP flow does not respond to congestion, the testbed network run in legacy mode allows the UDP flow to use nearly 25Gbps of bandwidth, starving the TCP flows. With EQDS (in trimming mode in this experiment), the UDP sender is throttled at the sending EQIF which enforces perfect ingress sharing, without any need for fair-queuing in the network.

Congestion controllers being nice. In fact, EQDS can enable **any** combination of existing congestion controllers to co-exist fairly without any in-network support. In figure 18 we show the sharing results when two senders send to the same receiver, with one sender using Cubic[17] and the other using the congestion control algorithms shown on the *x* axis label. We used all the congestion control implementations available in the Linux kernel, as well as the Mellanox DCQCN implementation. EQDS is able to fairly share the receiver's link without in-network support for almost all congestion controllers, in contrast to the status quo where latency-based schemes such as BBR[4] are starved by loss-based ones (e.g. Cubic). One exception are the Vegas[2] controllers that cannot utilize the 25Gbps link over EQDS when running alone, and that receive a smaller share when competing against Cubic. This appears to be due to Vegas measuring a small base RTT and stopping the window increase before it reaches a BDP.

The amount of buffering in the EQDS TX queue depends, as expected, on the congestion control algorithm. BBR is best, with an average latency of 243μs when sending at line rate, compared to DCTCP (K=16) at 435μs and Cubic at 1315μs.

RDMA versus TCP. To see how EQDS aids effective co-existence between different host stacks, we run a single TCP and *n* RDMA flows to the same destination in T2 (BlueField NICs). Figure 19 shows the bandwidth distribution among competing flows as the number of RDMA flows varies from zero to four. Without EQDS, TCP fills the switch buffers while DCQCN causes RDMA to back off, with some RDMA
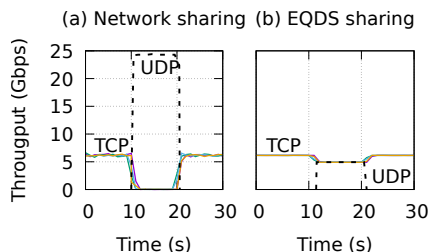
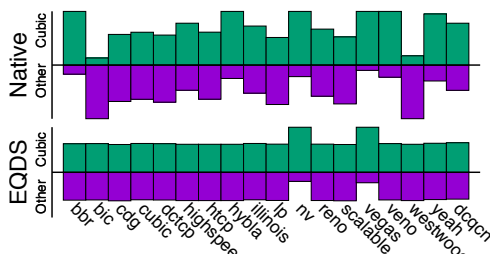Figure 17: Bandwidth sharing (5 to 1): one non-responsive UDP sender (testbed T1)

Figure 18: Capacity sharing between TCP variants, with and without EQDS (testbed T1)

Figure 19: RDMA and TCP network sharing (testbed T2)

flows being starved. With EQDS, data is buffered at the sending hosts and the bandwidth allocation is determined by the receiver, protecting RDMA and sharing the bandwidth fairly.

NVMeOF in parallel with a TCP shuffle. To understand sharing beyond a single bottleneck link, and to also test the CX4 RDMA stack over EQDS, we run the NVMeOF RDMA benchmark (§8.1) in parallel with TCP traffic emulating a MapReduce shuffle operation on the T2 testbed (CX4 hosts). Specifically, three nodes on one ToR run both iperf senders and NVMeOF targets. These three send to three nodes on another ToR running both iperf receivers and NVMeOF clients that perform random reads.

We observe that with EQDS the TCP shuffle alone is about 30% faster than without EQDS (26.1 Gbps vs 19.6 Gbps), in line with the previous experiments. When both TCP and RDMA run concurrently without EQDS, the shuffle throughput drops to 17.6 Gbps and NVMeOF drops to 2.56 Gbps – far from the optimal fair share. Under EQDS, shuffle and NVMeOF achieve 15.8 Gbps and 10 Gbps respectively, which is both a fairer allocation and higher overall throughput.

8.4 EQDS in legacy networks

To investigate EQDS with an oversubscribed core, we interconnect two ToRs in our T1 testbed with two 25Gbps spine links. Fair queuing is enabled. One ToR hosts three servers (2 at 100Gbps, 1 at 10Gbps); the other ToR hosts eight clients (mix of 10 and 25Gbps links). Each client connects to one server and continuously requests a 1MB object in a closed-loop, creating a new TCP connection each time. Clients are equally balanced across servers.

We increase the number of active clients from 1 (core utilization ≈50%) up to 8 (400% core over-subscription). We plot the median and 99% FCTs for baseline TCP, TCP-over-EQDS with trimming and TCP-over-EQDS using RTS. When the core is lightly loaded, EQDS's packet-level load balancing gives slightly smaller median (Figure 20) and 2x to 5x smaller 99th percentile FCT (Figure 21). As we add more clients and the core becomes overloaded, EQDS ends up behaving similarly to the native baseline, while RTS is slightly slower than baseline as it requires a large amount of buffering. For EQDS with trimming, as the core gets busier the trim rate increases and each packet can be re-sent multiple times. While this does
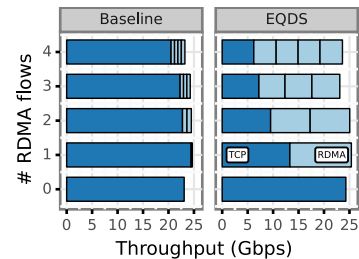
not affect FCT, it is undesirable; it would be better to reduce the pull rate when core congestion is detected.

Legacy "elephant" traffic. To understand how legacy traffic coexists with EQDS, we use the same configuration but with two clients downloading 1MB objects from two servers, and create a long-lasting iperf3 flow to another client. We vary the throughput for this legacy flow from 0 to 25Gbps and measure the FCTs of the other flows. Figure 22 shows how flow collisions on the spine between native 1MB flows and the elephant flow affects FCT: the median grows by 2.5x when the elephant flow fully occupies the spine link. With EQDS, packet spraying helps mitigate the effects of the elephant flow, and the increase in FCT is modest. A future implementation of load-aware routing should improve things further.

8.5 Host processing evaluation for EQDS

Our experiments so far have looked at how EQDS can help existing stacks, and used 25Gbps (our testbeds) and 10Gbps link speeds (EC2, simulation). We now examine the performance and overheads of our two EQDS implementations and evaluate how they perform at higher speeds. We tested using several configurations:

Setup 1. DPDK on the host, both with native transports and underneath the VM stack, where EQDS takes the role of the software switch used in virtualization.

Setup 2. DPDK on an SoC-based SmartNIC (Broadcom Stingray PS225 or Mellanox BlueField 2) with support for legacy TCP and RDMA traffic.

Setup 3. DPDK on the host, processing RDMA traffic to and from a Mellanox CX4 NIC.

Setup 4. Our Linux Kernel 5.4 implementation, with EQDS kernel module running underneath the TCP/IP stack.

The DPDK implementation is the most versatile and performs best, though this depends on the higher level stack and the setup used (Figure 23). The best performance is given by `eqdsperf` in setup 1 using the zero-copy native EQDS transport. Between two 2.5GHz Xeon Silver 4215 machines with Intel Columbiaville NICs, eqdsperf achieves 40Gbps with 1.5KB packets and 100Gbps with 4KB packets. The bottleneck is the sender; with two senders to the same receiver, the link saturates with 3KB packets.
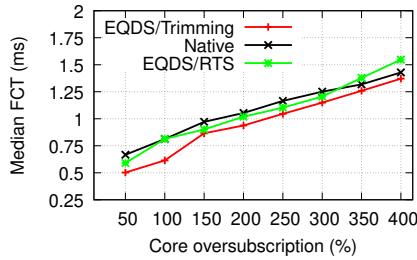
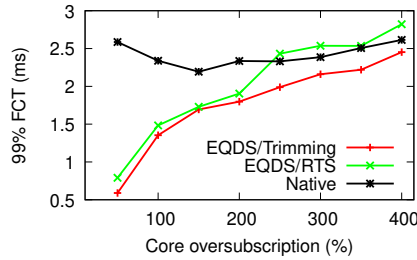Figure 20: Median FCT for 1MB flows (oversubscribed core).

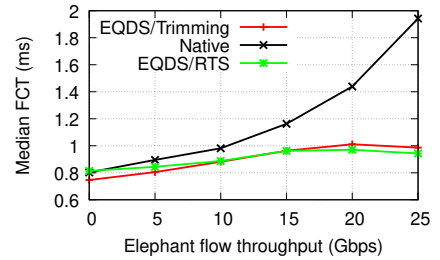Figure 21: 99% FCT for 1MB flows (oversubscribed core).

Figure 22: Median FCT with an elephant legacy flow.

Setup	Higher stack	Throughput (Gbps) Link	1.5KB/9KB	Latency ping(us)
(1) Host	eqdsperf	100	40/98	15(+1)
(1) Host	Linux VM	100(40)	27/55	18(+1)
(2)PS225	Linux	25	23.6/24.1	31(+12)
(2)PS225	RDMA	25	23.5/24.1	18(+12)
(2)Blue2	Linux	100	23/49	22(+16)
(2)Blue2	RDMA	100	23/49	22(+16)
(3)Host/CX4	RDMA	10	9.6/9.6	13(+9)

Figure 23: EQDS DPDK performance
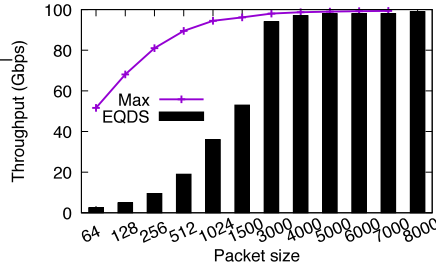
Figure 24: `eqdsperf` throughput (setup 1)

Stack	MTU	TSO off GRO off	TSO on GRO on
Linux	*1.5KB*	17Gbps	40Gbps
EQDS	*1.5KB*	5Gbps	27Gbps
Linux	*9KB*	40Gbps	60Gbps
EQDS	*9KB*	28Gbps	55Gbps

Figure 25: EQDS Kernel performance (setup 4)

In Setup 1, when we run EQDS underneath Linux VMs in KVM using vhost-user networking, it achieves 27Gbps with 1.5KB packets compared to 40Gbps for the baseline (testpmd). The cost here for both testpmd and EQDS comes from the packet copy from the guest to the host.

In Setup 2 we run EQDS on both the Broadcom PS225 SmartNIC and Mellanox's BlueField 2 NIC. Our EQIF uses one NIC core for each of the two links and one for credit pacing. On the Stingray, EQDS saturates the link with a single core when the MTU is 1.5KB or larger. On the PS225 the results are similar with EQDS reaching 50Gbps with an 8KB MTU. We note that SmartNIC ARM cores are weaker than x86 cores and memory bandwidth seems limited; reaching 100Gbps may require more offloading.

In Setup 3 we divert RDMA traffic from a Mellanox ConnectX-4 Lx NIC via a host core. This is not ideal as it requires an extra round trip across the PCIe bus, but demonstrates RDMA-over-EQDS even without a smart NIC. Since the CX4 PCIe bandwidth is 56Gbps, we limit the link bandwidth to 10Gbps to avoid a PCIe bottleneck. With EQDS, the bandwidth of a single RDMA flow between two RDMA nodes remains the same as with baseline RDMA (Table 23), but the additional PCIe round-trip increases median latency by 9 μs on an unloaded system and by 14 μs under load.

In Setup 4 we run EQDS in a kernel module. Figure 25 shows TCP performance with and without EQDS, and explores the effect of TCP offloads. Our kernel implementation can reach 55Gbps (one core) with jumbograms, or 27Gbps with a 1.5KB MTU. TCP's dependence on offload support is clear. EQDS increases CPU utilization at the sender by 2% and at the receiver by 5% when running an iperf at 55Gbps.

Tunnel setup overhead. When EQDS tunnels are already set up, EQDS only adds a few *μs* of latency, as shown in Fig. 23. When a new host is contacted, however, a tunnel is setup dynamically using a zero-RTT approach. We measure this by tearing down tunnels, then sending a series of pings; the difference between the first ping time and subsequent ones is around 20μs. This setup latency is due to memory initialization costs for the tunnel data structures.

EQDS memory consumption. How does EQDS memory consumption scale with the size of the datacenter? There are three categories of memory to consider.

A Sending EQIF buffers packets awaiting transmission. For TCP, we use 100 packets per destination by default, but we find that if needed this can be reduced as low as four packets per destination when sending to many receivers. For RDMA, the upper marking threshold is set to 375 packets.

The receiving EQIF has a reorder buffer per sender. In the worst case with faulty links, the buffer can reach the EQDS window which is between 50 packets (DPDK) and 150 packets (kernel). In practice this is limited to less than 10 packets.

In-flight packets are buffered at the sender pending retransmission, but the pending buffer size is limited by the BDP. We verify this by starting iperfs to an increasingly larger number of receivers. The total number of in-flight packets across all tunnels saturates at around 400 packets (NIC ring size plus one BDP) regardless of the number of receivers.

With 1.5GB of DRAM, EQDS can buffer packets for 10,000 active destination; this fits in the RAM of the Smart-NICs we used (8-16GB) as well as the hosts. In practice both the number of active destinations and actual buffer utilization are smaller; on EC2 the receiver's memory usage for EQDS did not exceed 100MB with 850 senders.

9 Related work

Tunneling is widely used in datacenters for security isolation of traffic (e.g. VXLAN and GRE). These protocols, however, do not offer any performance isolation between tenants. Rate limiting is typically used in public clouds, but this offers limited isolation, especially for incast workloads.

Numerous research works examined the performance sharing problem [40, 34, 35]. Seawall [40] and ElasticSwitch [35] use rate-limiting between each pair of hosts in a datacenter to manage sharing, and use centrally computed weights to achieve fair sharing among different tenants. FairCloud [34] discusses fundamental tradeoffs in sharing cloud networks. This line of works relies on rate limiting and needs large in-network queues to cope with incast; it also doesn't improve utilization of multipath networks.

Virtualized congestion control works such as VCC [6] and AC/DC [19] propose ways of deploying new TCP congestion variants without VM changes. Both keep per-flow congestion state in the hypervisor, applying rate-limiting techniques (such as receive window reduction) to force the VM stack to reduce its rate; both show how DCTCP can be deployed in this way. OnRamp [3] is a recent proposal that aims to improve the fast start phase of transport protocols by tracking fine-grained RTT measurements per flow, in the hypervisor, and then stopping packets from entering the network when latency increases above a certain threshold. This line of work does not allow multipath transmission of TCP packets or other network-specific enhancements, and still requires large in-network buffers to cope with incast.

EQDS takes the next logical step over this line of prior work: it completely decouples host stacks from the network via edge queues, thus supporting multiple higher layer transports (e.g. TCP, RDMA or native). EQDS does not do rate limiting (because it can build large queues), but uses a receiver-driven control loop instead, ensuring that network queue depths are kept as low as possible. Any sharing outcome for clouds (e.g., [34]) can be configured using the primitives EQDS provides to higher layers. The key benefit of EQDS is decoupling the higher layer transports from lower layer implementations; this enables regular TCP run over packet-sprayed networks, among others.

Fastpass [33], pHost [14], Aeolus [20], NDP [18], Homa [31], qJump [15] are techniques that show how one can operate a network at high load and small queues. In-band network telemetry [32] provides accurate congestion information to endpoints that can be used in over-subscribed topologies (e.g. HPCC [25]). In contrast to all these transport or congestion control protocols, EQDS is a congestion tunneling layer that is meant to allow other transports to operate on top; it can use the mechanisms proposed by any of these or other, yet-to-be-invented, mechanisms to drive packet pacing and ensure that the network core is used efficiently, but it also allows to deploy them transparently to user applications. Our implementation implements both NDP [18] and a request-to-send

variant that does not require trimming.

New transport stacks like PonyExpress [27] or eRPC [23] show the benefits of kernel bypass to support novel APIs, but they face an uphill battle in deployment. EQDS allows such transports to be deployed without changes to the network.

10 Conclusions and next steps

EQDS is a new network layer that provides strong performance isolation among co-existing transports by pulling the shared queues out of the data center network core and moving them to the edge. More importantly, it introduces a new datagram service abstraction which fully decouples the transport services above it from the network implementation underneath, thus enabling independent evolution of these layers while ensuring future compatibility among them.

EQDS is designed for gradual adoption. From the application perspective, RDMA and TCP EQIFs allow seamless deployment of EQDS without any application modifications. In the longer term we envision the emergence of application protocols implemented atop and optimized for EQDS-native APIs, thereby taking full advantage of the improved visibility into the network performance that they offer.

From the network infrastructure perspective, one can use our software-only implementations in Linux kernel, Smart-NIC or/and host that allow the benefits of EQDS isolation to be reaped at low performance cost, and can also run without switch support for packet trimming.

The NIC implementation is ideal from a deployment point of view, as it is cleanly separated from the host software, which can use offloading support as before. Our smart NIC setup, however, adds a little unnecessary latency (12-16us) and is limited to 25Gbps per core at 1.5KB MTU. These limitations should be fixable with NIC ASIC support for EQDS.

There are many examples of existing ASIC and FPGA implementations of connected transport protocols, ranging from RDMA [9, 21, 22, 41], to TCP offload engines [30, 12, 39]. An EQDS NIC would build upon these works, and its hardware version might actually be simpler: keeping state for each connected endpoint rather than per-flow means more connections could be stored in on-chip memory, and its relaxed ordering and reliability guarantees allow implementing a simpler state machine. By offloading control packet processing, we expect to shorten the EQDS control loop delay significantly, and reducing its jitter, thus minimizing switch buffer usage. Furthermore, offloading connection setup and teardown logic would allow low latency small flows to get close to native performance.

References

[1] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. "Data Center TCP (DCTCP)". In: *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2010.

[2] L.S. Brakmo and L.L. Peterson. "TCP Vegas: end to end congestion avoidance on a global Internet". In: *IEEE Journal on Selected Areas in Communications* 13.8 (1995), pp. 1465–1480.

[3] "Breaking the Transience-Equilibrium Nexus: A New Approach to Datacenter Packet Transport". In: *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, Apr. 2021.

[4] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. "BBR: Congestion-Based Congestion Control". In: *ACM Queue* 14, September-October (2016), pp. 20–53.

[5] Yanpei Chen, Rean Griffith, Junda Liu, Randy H. Katz, and Anthony D. Joseph. "Understanding TCP Incast Throughput Collapse in Datacenter Networks". In: *Workshop on Research on Enterprise Networking (WREN)*. ACM, 2009.

[6] Bryce Cronkite-Ratcliff, Aran Bergman, Shay Vargaftik, Madhusudhan Ravi, Nick McKeown, Ittai Abraham, and Isaac Keslassy. "Virtualized Congestion Control". In: *SIGCOMM '16*. Association for Computing Machinery, 2016, pp. 230–243.

[7] Steve Deering. *Watching the waist of the internet hourglass*. ICNP plenary. 1998.

[8] DPDK. *DPDK Porgrammer's Guide » Switch Representation within DPDK Applications*. 2019. URL: `https://doc.dpdk.org/guides-19.11/prog_guide/switch_representation.html`.

[9] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. M. Merritt, E. Gronke, and C. Dodd. "The Virtual Interface Architecture". In: *IEEE Micro* 18.2 (Mar. 1998), pp. 66–76.

[10] Haggai Eran. *libconntrack-cm – Connection Tracking for RDMA CM*. 2022. URL: `https://github.com/acsl-technion/libconntrack-cm`.

[11] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. "Hedera: Dynamic Flow Scheduling for Data Center Networks". In: *Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2010.

[12] W. Feng, P. Balaji, C. Baron, L.N. Bhuyan, and D.K. Panda. "Performance characterization of a 10-Gigabit Ethernet TOE". In: *13th Symposium on High Performance Interconnects (HOTI'05)*. 2005, pp. 58–63.

[13] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayantara Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Yuan He, and Christina Delimitrou. "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud and Edge Systems". In: *Proceedings of the Twenty Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Apr. 2019.

[14] Peter X. Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. "pHost: Distributed Near-optimal Datacenter Transport over Commodity Network Fabric". In: *Conference on Emerging Networking Experiments and Technologies (CoNEXT)*. ACM, 2015.

[15] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. "Queues Don't Matter When You Can JUMP Them!" In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, May 2015, pp. 1–14.

[16] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. "RDMA over Commodity Ethernet at Scale". In: *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2016.

[17] Sangtae Ha, Injong Rhee, and Lisong Xu. "CUBIC: A New TCP-Friendly High-Speed TCP Variant". In: *SIGOPS Oper. Syst. Rev.* 42.5 (July 2008), pp. 64–74.

[18] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. "Re-architecting Datacenter Networks and Stacks for Low Latency and High Performance". In: *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2017.

[19] Keqiang He, Eric Rozner, Kanak Agarwal, Yu (Jason) Gu, Wes Felter, John Carter, and Aditya Akella. "AC/DC TCP: Virtual Congestion Control Enforcement for Datacenter Networks". In: *SIGCOMM '16*. Association for Computing Machinery, 2016, pp. 244–257.

[20] Shuihai Hu, Wei Bai, Gaoxiong Zeng, Zilong Wang, Baochen Qiao, Kai Chen, Kun Tan, and Yi Wang. "Aeolus: A Building Block for Proactive Transport in Datacenters". In: *SIGCOMM '20*. Association for Computing Machinery, 2020, pp. 422–434.

[21] InfiniBand Trade Association (IBTA). *About Infini-Band*. (Accessed: May 2021). URL: https://www.infinibandta.org/about-infiniband/.

[22] InfiniBand Trade Association (IBTA). *The RoCE Initiative*. (Accessed: May 2021). URL: https://www.infinibandta.org/roce-initiative/.

[23] Anuj Kalia, Michael Kaminsky, and David G. Andersen. "Datacenter RPCs Can Be General and Fast". In: *NSDI'19*. USENIX Association, 2019, pp. 1–16.

[24] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. "Swift: Delay is Simple and Effective for Congestion Control in the Datacenter". In: *SIGCOMM '20*. Association for Computing Machinery, 2020, pp. 514–528.

[25] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. "HPCC: High Precision Congestion Control". In: *SIGCOMM '19*. Association for Computing Machinery, 2019, pp. 44–58.

[26] Yuanwei Lu, Guo Chen, Bojie Li, Kun Tan, Yongqiang Xiong, Peng Cheng, Jiansong Zhang, Enhong Chen, and Thomas Moscibroda. "Multi-Path Transport for RDMA in Datacenters". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Apr. 2018, pp. 357–371.

[27] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. "Snap: A Microkernel Approach to Host Networking". In: *SOSP '19*. Association for Computing Machinery, 2019, pp. 399–413.

[28] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. "TIMELY: RTT-Based Congestion Control for the Datacenter". In: *SIGCOMM '15*. Association for Computing Machinery, 2015, pp. 537–550.

[29] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. "Revisiting Network Support for RDMA". In: *SIGCOMM '18*. Association for Computing Machinery, 2018, pp. 313–326.

[30] Jeffrey C. Mogul. "TCP Offload is a Dumb Idea Whose Time Has Come". In: *HOTOS'03*. USENIX Association, 2003, pp. 5–5.

[31] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. "Homa: A Receiver-driven Low-latency Transport Protocol Using Network Priorities". In: *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2018.

[32] p4.org Applications Working Group. *In-band Network Telemetry (INT) Dataplane Specification*. 2020. URL: https://github.com/p4lang/p4-applications/blob/master/docs/INT_v2_1.pdf.

[33] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. "Fastpass: A Centralized "Zero-queue" Datacenter Network". In: *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2014.

[34] Lucian Popa, Gautam Kumar, Mosharaf Chowdhury, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. "FairCloud: Sharing the Network in Cloud Computing". In: *SIGCOMM '12*. Association for Computing Machinery, 2012, pp. 187–198.

[35] Lucian Popa, Praveen Yalagandula, Sujata Banerjee, Jeffrey C. Mogul, Yoshio Turner, and Jose Renato Santos. "ElasticSwitch: Practical Work-Conserving Bandwidth Guarantees for Cloud Computing". In: *SIGCOMM '13*. Association for Computing Machinery, 2013, pp. 351–362.

[36] R. Ludwig and A. Gurtov. *RFC4015: The Eifel Response Algorithm for TCP*. 2003. URL: https://tools.ietf.org/html/rfc4015.

[37] R. Ludwig and M. Meyer. *RFC3522: The Eifel Detection Algorithm for TCP*. 2003. URL: https://tools.ietf.org/html/rfc3522.

[38] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. "Improving Datacenter Performance and Robustness with Multipath TCP". In: *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2010.

[39] Mario Ruiz, David Sidler, Gustavo Sutter, Gustavo Alonso, and Sergio López-Buedo. "Limago: An FPGA-Based Open-Source 100 GbE TCP/IP Stack". In: *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. 2019, pp. 286–292.

[40] Alan Shieh, Srikanth Kandula, Albert Greenberg, Changhoon Kim, and Bikas Saha. "Sharing the Data Center Network". In: *NSDI'11*. USENIX Association, 2011, pp. 309–322.

[41] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulka-rni, and Gustavo Alonso. "StRoM: Smart Remote Memory". In: *EuroSys '20*. Association for Computing Machinery, 2020.

[42] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F. Wenisch, Monica Wong-Chan, Sean Clark, Milo M. K. Martin, Moray McLaren, Prashant Chandra, Rob Cauble, Hassan M. G. Wassel, Behnam Montazeri, Simon L. Sabato, Joel Scherpelz, and Amin Vahdat. "1RMA: Re-Envisioning Remote Memory Access for Multi-Tenant Datacenters". In: *SIGCOMM '20*. Association for Computing Machinery, 2020, pp. 708–721.

[43] Anirudh Sivaraman, Suvinay Subramanian, Moham-mad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. "Programmable Packet Scheduling at Line Rate". In: *SIGCOMM '16*. Association for Computing Machinery, 2016, pp. 44–57.

[44] Haitao Wu, Zhenqian Feng, Chuanxiong Guo, and Yongguang Zhang. "ICTCP: Incast Congestion Control for TCP in Data-Center Networks". In: *IEEE/ACM Trans. Netw.* 21.2 (Apr. 2013), pp. 345–358.

Appendix A. Protocol state machines

The state machines for our protocol implementation are shown in Figures 26 (the sender) and 27 (for the receiver).

The aim is for the two endpoints to self-synchronize. The sender sets the SYN flag on all data packets sent until it receives a SYN-ACK (or SYN-NACK or SYN-SACK) packet from the receiver. As a window of SYN packets can arrive out-of-order, the lowest ten bits of the sequence number are always zero in the first SYN packet and the upper sequence number bits in SYN packets indicate an epoch number.

The receiver state machine is very simple, with just two states, `closed` and `established`. As SYNs from the initial window can arrive out of order, an arriving SYN causes the receiver to move to established state and set the initial sequence number to be that from the SYN with the lowest ten bits cleared. Any subsequent SYNs with the same epoch number are treated as normal data packets except the SYN flag is set in their ACKs.

When data arrives at the sender it chooses a random epoch number, starts sending data with SYN set, and moves to SYN-SENT state. It then moves to *established* state on receiving a SYN-ACK with the correct epoch number from the receiver.

The epoch number is needed because either end can drop state at any point. If a sender drops state very early in a connection, then immediately tries to re-establish a tunnel, old SYN packets may still be in flight. The epoch number allows the sender and receiver to agree which is the new connection.

If the sender retains the old epoch number, it simply chooses a greater epoch for the new connection. The receiver then accepts the new SYN as re-establishing the connection seamlessly. However, if the sender has no state, it chooses a random epoch number. If this random epoch number is greater it will be accepted, but if it is lesser, the receiver concludes it is old and replies with a SYN-ACK advertising its current epoch. If the sender receives such a mismatched greater epoch in a SYN-ACK, it concludes the setup has failed, chooses a new epoch greater than that advertised by the receiver, and resends its initial window of SYNs. This new attempt will then always succeed.

It is possible that SYN-ACKs from the previous connection are still in flight when the new connection is attempted. If these have a lower epoch (the common case) they are ignored; if they have a greater epoch they trigger the re-sync process as described above.

If a receiver has outstanding data in its reorder queue when a connected reestablishes, it releases this data to the host out-of-order as it can no longer guarantee the sequence space holes will be filled. This is expected to be very rare in practice as endpoints will not normally drop state with unacked data in transit.

The sender responds to any indication of an unhealthy tunnel by closing it and re-establishing a new tunnel, possibly after a timeout. A large number of retransmit timeouts implies that there is likely a network connectivity issue affecting the tunnel. Similarly, receipt of a RST indicates the receiver is in the `closed` state, and prompts re-establishment of the tunnel if there are packets in the EQIF TX queue.

The key benefit of this self-synchronizing design is that either endpoint can drop state without reliably informing its peer. This gives EQDS implementations a lot of freedom in managing per-tunnel memory, allowing lightly used tunnels to be dropped in memory pressure scenarios. While these ability is not used in our software implementations, we expect that future ASIC implementations of EQDS in NICs will make full use of these features.

Appendix B. Details on the experiment setup.

Our T1 (TCP/IP) testbed has 10 endpoints:

- 8 Linux kernel endpoints emulated on four servers with Intel Xeon Silver 4215 CPUs @ 2.50GHz (8 cores, 16 hyperthreads), 128GB of RAM and a dual-port 100Gbps Intel Columbiaville NIC each (running in Setup 1, with EQDS on the host cores either as part of the kernel or as a DPDK process).

- 4 Broadcom endpoints emulated by two servers with Intel Xeo E5-2650L v2 CPUs @ 1.70GHz, 32GB of RAM (10 cores, 20 hyperthreads), each with a dual-port 25Gbps
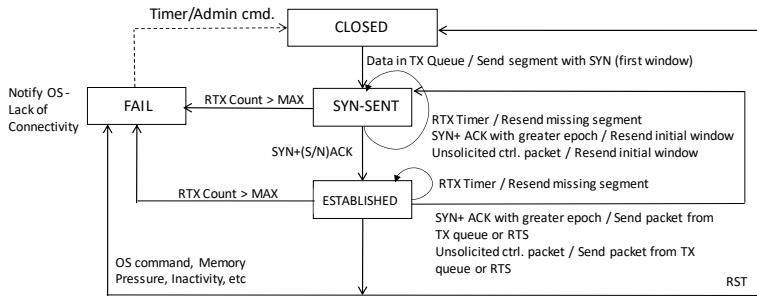
Figure 26: EQDS sender state machine

Figure 27: EQDS receiver state machine

Broadcom Stingray NIC (EQDS is running on the Arm cores of the Stingray, in Setup 2).

Each endpoint is connected via a 25Gbps link to a 64-port Tofino 1 switch; we downgraded the Columbiaville NICs to 25Gbps to match the speed of the Stingray NICs. We implemented packet trimming in the switch via a combination of ingress meters and cloning sessions. We used the four pipelines in this switch to emulate four ToR switches; these are connected via DAC cables to two spine switches emulated by another 32-port Tofino (one pipeline per emulated switch). Switch buffers are set to 15 packets (125KB) for EQDS and 200 packets (2MB) for TCP. Cross-sectional bandwidth is 200Gbps (slight over-subscription).

We run the workload in our Linux machines with and without EQDS, and measure performance using `ping`, `iperf` and `iperf3` for UDP tests. Our kernel and DPDK stacks interoperate and at 25Gbps have similar performance, so we omit a performance breakdown.

Our T2 (RDMA) testbed has six endpoints, each with a 2.1GHz Intel Xeon E5-2620 v2 CPU. The hosts are connected to an NVIDIA Spectrum SN3700 switch, configured using loopback cables as a 2-tier Clos topology with 40 Gbps bisectional bandwidth. All links are configured to 10 Gbps with 4 KiB MTU. Each of six ports of the switch is connected either to one of three dual-port 8-core NVIDIA BlueField-2 Smart NIC (two of which clock at 2.5 GHz and one at 2 GHz), or to an NVIDIA Mellanox ConnectX-4 Lx.

We implement trimming in the SN3700 by mirroring and truncating dropped frames, then sending them to a dedicated loopback port where a P4 program redirects them back to the destination port and appropriate queue. When trimming is enabled, switch buffers are set to 60 kB.

When using ConnectX-4 Lx NICs, EQDS runs on the host CPU instead of the SmartNIC, and traffic between the host network stack and EQDS uses NIC loopback.

The baseline RDMA performance for SmartNICs is measured by configuring the SmartNICs to forward traffic between the host and the network in hardware, without going through the ARM cores. For NICs, the host network stack uses the RDMA NIC directly.
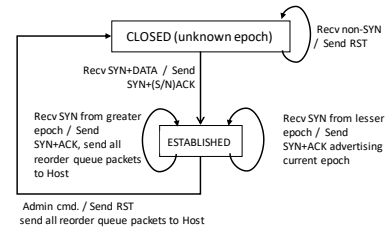
Buffer settings for TCP EQIFs. Our experiments show that TCP/IP runs smoothly over EQDS with our default settings (sending EQIFs buffer up to 100 packets per destination), but does this change with other buffer sizes? Larger buffers simply result in more EQDS sender-side buffering for TCP Cubic, and do not affect performance at all. What about smaller buffers?

When the send buffer is at least one BDP (30 packets in our testbed), all TCP variants achieve line-rate; below that the throughput depends on the congestion controller. DCTCP maintains full utilization with K=16, while BBR needs half that to reach line rate..

DCQCN parameter settings for the RDMA EQIF. Our RDMA EQIF performs flow control using the NIC's DC-QCN implementation, but it has a shorter control loop than a baseline DCQCN setup: it sends CNPs directly to the sender NIC instead of simply marking packets and waiting for the receiver to send CNPs. This allows more aggressive DCQCN parameters to be used.

We explored the DCQCN parameter space, varying the EQIF's probabilistic marking thresholds (K_{low}/K_{high}), active increase rate (R_{AI}) and rate increase/decrease timers (R_d/R_i). Our goal was to be able to stop the RDMA sender quickly during large incasts (1Mbps per sender) without dropping packets, and to be able to resume at line rate (i.e. have enough buffering) when an incast subsides (available rate is 25Gbps). The resulting parameters, shown below, are as aggressive as possible without under or overflowing the EQIF queue.

	K_{low}	K_{high}	R_d	R_i	R_{AI}
Baseline	150 kB	1500 kB	4 μsec	300 μsec	5 Mbps
EQDS(BF2)	72 kB	584 kB	4 μsec	750 μsec	5 Mbps
EQDS(Cx4)	150 kB	1500 kB	4 μsec	128 μsec	50 Mbps

Backpressure Flow Control

Prateesh Goyal[1], Preey Shah[2], Kevin Zhao[3], Georgios Nikolaidis[4],
Mohammad Alizadeh[1], Thomas E. Anderson[3]

[1]MIT CSAIL, [2]IIT Bombay, [3]University of Washington, [4]Intel, Barefoot Switch Division

Abstract

Effective congestion control for data center networks is becoming increasingly challenging with a growing amount of latency-sensitive traffic, much fatter links, and extremely bursty traffic. Widely deployed algorithms, such as DCTCP and DCQCN, are still far from optimal in many plausible scenarios, particularly for tail latency. Many operators compensate by running their networks at low average utilization, dramatically increasing costs.

In this paper, we argue that we have reached the practical limits of end-to-end congestion control. Instead, we propose, implement, and evaluate a new congestion control architecture called *Backpressure Flow Control* (BFC). BFC provides per-hop per-flow flow control, but with bounded state, constant-time switch operations, and careful use of buffers and queues. We demonstrate BFC's feasibility by implementing it on Tofino2, a state-of-the-art P4-based programmable hardware switch. In simulation, we show that BFC achieves near optimal throughput and tail latency behavior even under challenging conditions such as high network load and incast cross traffic. Compared to deployed end-to-end schemes, BFC achieves 2.3 - 60× lower tail latency for short flows and 1.6 - 5× better average completion time for long flows.

1 INTRODUCTION

Single and multi-tenant data centers have become one of the largest and fastest growing segments of the computer industry. Data centers are increasingly dominating the market for all types of high-end computing, including enterprise services, parallel computing, large scale data analysis, fault-tolerant middleboxes, and global distributed applications [10, 27, 47]. These workloads place enormous pressure on the data center network to deliver, at low cost, ever faster throughput with low tail latency even for highly bursty traffic [24, 63].

Although details vary, almost all data center networks today use a combination of endpoint congestion control, FIFO queues at switches, and end-to-end feedback of congestion signals like delay or explicit switch state to the endpoint control loop.[1] As link speeds continue to increase, however, the design of the control loop becomes more difficult. First, more traffic fits within a single round trip, making it more

difficult to use feedback effectively. Second, traffic becomes increasingly bursty, so that network load is not a stable property except over very short time scales. And third, switch buffer capacity is not keeping up with increasing link speeds (Fig. 1), making it even more challenging to handle traffic bursts. Most network operators run their networks at very low average load, throttle long flows at far below network capacity, and even then see significant congestion loss.

Instead, we propose a different approach. The key challenge for data center networks, in our view, is to efficiently allocate buffer space at congested network switches. This becomes easier and simpler when control actions are taken per flow and per hop, rather than end-to-end. Despite its advantages, per-hop per-flow flow control appears to require per-flow state at each switch, even for quiescent flows [11,41], something that is not practical at data center scale.

Our primary contribution is to show that per-hop per-flow flow control can be *approximated* with a limited amount of switch state and modest number of switch queues, using only simple constant-time switch operations on a modern programmable switch. Instead of all flows, we only need state and dedicated queues for *active flows*—those flows with queued packets. We show that, with switch-level fair queueing or shortest flow scheduling, the number of active flows is modest for typical data center workloads, even in the tail of the distribution. The tradeoff is that performance can degrade when the number of active flows exceeds the number of queues. In practice, we advocate combining per-hop flow control with end-to-end congestion control to avoid pathological behavior. However, to better illustrate the benefits and limitations of our approach, our description and experiments focus on comparing pure per-hop control with pure end-to-end control.

We have implemented our approach, *Backpressure Flow Control (BFC)*, on Tofino2 a state-of-the-art P4-based programmable switch ASIC supporting 12.8 Tbps of switching capacity [33]. Tofino2 has 32-128 independently pausable queues at each output port. Our implementation uses less than 10% of the dedicated stateful memory on Tofino2. All per-packet operations are implemented entirely in the dataplane; BFC runs at full switch capacity.

To evaluate performance, we run large-scale ns-3 [4] simulations using synthetic traces drawn to be consistent with measured workloads from Google and Facebook data centers [49] on an oversubscribed multi-level Clos network

---

[1]We refer to schemes that rely on feedback signals delayed by an entire round-trip-time as *end-to-end* schemes, to contrast them with hop-by-hop mechanisms.

topology. We synthetically add incast to these workloads to represent a challenging scenario for both end-to-end and per-hop approaches. We consider both throughput and tail latency performance for short, medium, and long flows.

For our simulated workloads, BFC improves both latency for short flows and throughput for long flows. Compared to a wide set of deployed end-to-end systems, including DCTCP [8], DCQCN [65], and HPCC [43], BFC achieves 2.3 - 60× better tail flow completion times (FCTs) for short flows, and 1.6 - 5× better average performance for long flows. ExpressPass [22] achieves 35% better short flow tail latency, but 17× worse average case performance for long flows. We also show that BFC performs close to an idealized fair queueing system with unbounded buffers and switch queues, but with limited queues and far smaller buffers. BFC can be combined with other switch scheduling algorithms such as priority scheduling among traffic classes. Unlike other receiver-driven schemes like Homa [49], BFC does not assume knowledge of flow sizes and does not rely on packet spraying (which is difficult to deploy in practice). With packet spraying, Homa outperforms BFC, but without it we show BFC outperforms Homa and can enforce shortest remaining flow first scheduling more accurately.

Our specific contributions are:

- A discussion of the fundamental limits of end-to-end congestion control for high bandwidth data center networks.
- A practical protocol for per-hop per-flow flow control, called BFC, that uses a small, constant amount of state and limited number of switch queues to achieve near-optimal tail-latency performance for typical data center workloads.
- An implementation and proof-of-concept evaluation of BFC on a commercial switch. To our knowledge, this is the first implementation of a per-hop per-flow flow control scheme for a multi-Tbps switch.

2 MOTIVATION

Over the last decade, researchers and data center operators have proposed a variety of congestion control algorithms for data centers, including DCTCP [8], Timely [48], Swift [40], DCQCN [65], and HPCC [43]. The primary goals of these protocols are to achieve high throughput, low tail packet delay, and high resilience to bursts and incast traffic patterns. Operationally, these protocols rely on *end-to-end* feedback loops, with senders adjusting their rates based on congestion feedback signals echoed by the receivers. Irrespective of the type of signal (e.g., ECN marks, multi-bit INT information [36, 43], delay), the feedback delay for these schemes is a network round-trip time (RTT). This delay has an important role in the performance of end-to-end schemes. In particular, senders require at least one RTT to obtain feedback, and therefore face a hard tradeoff in deciding the starting rate of a flow. They can either start at a high rate and risk causing congestion, or start at a low rate and risk
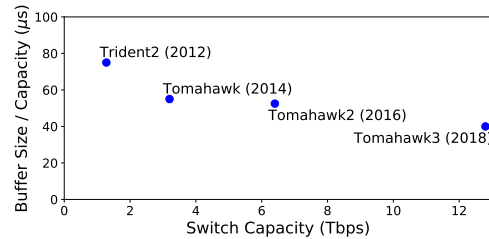
Figure 1: Hardware trends for top-of-the-line data center switches from Broadcom. Switch capacity and link speed have been growing rapidly, but buffer size is not keeping up with increases in switch capacity.

under-utilizing the network. Moreover, even after receiving feedback, senders can struggle to determine the right rate if the state of the network (e.g., link utilization and queuing delay) changes quickly compared to the RTT.

We argue that three trends are making these problems worse over time, and will make it increasingly difficult to achieve good performance with end-to-end protocols.

Trend 1: Rapidly increasing link speed. Fig. 1 shows the switch capacity of top-of-the-line data center switches manufactured by Broadcom [20, 50, 61]. Switch capacity and link speeds have increased by a factor of 10 over the past six years with no signs of stopping.

Trend 2: Most flows are short. Fig. 2 shows the byte-weighted cumulative distribution of flow sizes for three industry data center workloads [49]: (1) All applications in a Google data center, (2) Hadoop cluster in a Facebook center, and (3) a WebSearch workload. Each point is the fraction of all bytes sent that belong to flows smaller than a threshold for that workload. For example, for the Google workload, flows that are shorter than 100 KB represent nearly half of all bytes. As link speed increases, a growing fraction of traffic belongs to flows that complete quickly relative to the RTT. For example, most Facebook Hadoop traffic is likely to fit within one round trip within the next decade. While some have argued that data center flows are increasing in size [6], the trend is arguably in the opposite direction with the growing use of RDMA for fine-grained remote memory access.

Trend 3: Buffer size is not scaling with switch capacity. Fig. 1 shows that the total switch buffer size relative to its capacity has decreased by almost a factor of 2 (from $75\,\mu s$ to $40\,\mu s$) over the past six years. With smaller buffers relative to link speed, buffers now fill up more quickly, making it more difficult for end-to-end congestion control to manage those buffers.

2.1 Limits of End-to-End Congestion Control

This combination — very fast links, short flows, and inadequate buffers — creates the perfect storm for end-to-end congestion control protocols. Flows that complete within one or a few RTTs (which constitute an increasingly larger fraction of traffic) either receive no feedback, or last for so few feedback cycles that they cannot find the correct
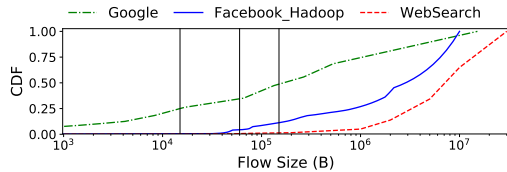
Figure 2: Cumulative bytes contributed by different flow sizes for three different industry workloads. The three vertical lines show the BDP for a 10 Gbps, 40 Gbps, and 100 Gbps network, assuming a $12\,\mu s$ RTT.
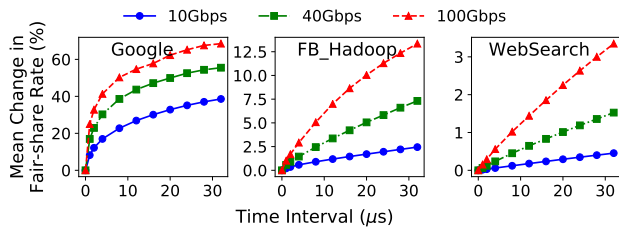

Figure 3: Mean percent change in fair-share rate as a function of workload, delay, and bandwidth.

Scheme	Throughput (%)	99% Queuing Delay (μs)
BFC	37.3	1.2
HPCC	22.9	23.9
DCQCN	10.0	30.4

Table 1: For a shared 100 Gbps link, BFC achieves close to ideal throughput (40%) for the long flow, with low tail queuing delay.

rate [34]. For longer flows, the rapid arrival and departure of cross-traffic creates significant fluctuations in available bandwidth at RTT timescales, making it difficult to find the correct rate. The result is loss of throughput and large queue buildup. Insufficient switch buffering further exacerbates these problems, leading to packet drops or link-level pause events (PFC [62]) that spread congestion upstream.

To understand these issues, we consider an experiment with a long-lived flow competing on a single link against cross-traffic derived from the Google, Facebook, and WebSearch workloads. We repeat the experiment at 10, 40, and 100 Gbps, with the average load of the cross-traffic flows set to be 60% of the link capacity in each case. Fig. 3 plots the relative change in the fair-share rate of the long-lived flow over different time intervals.[2] Congestion control protocols struggle to track the fair-share rate when it varies significantly over their feedback delay (typically an RTT). As link speeds increase or flows become shorter, the fair-share rate changes more rapidly (since flows arrive and finish more quickly), and hence congestion control becomes more difficult.

Table 1 considers one configuration in detail, with a single long flow sharing a 100 Gbps link with cross-traffic drawn from the Facebook distribution at 60% average load. The

---

[2]The fair-share rate ($f(t)$) for a link of capacity C shared by $N(t)$ flows is $C/N(t)$. The relative change in $f(t)$ over time interval I is given by $\left| \frac{f(t+I)-f(t)}{f(t)} \right|$.

minimum RTT (hence, feedback delay) is $8\,\mu s$. We consider both the single packet (99th percentile) queuing delay and throughput for the long flow, for our approach (BFC) and two end-to-end protocols (DCQCN and HPCC). BFC is able to achieve close to the maximum possible throughput for the long-lived flow (40%) with low tail delay, while the end-to-end protocols fall short in both respects.

2.2 Existing Solutions are Insufficient

There are several existing solutions that go beyond end-to-end congestion control. We briefly discuss the most prominent of these approaches and why they are insufficient to deal with the challenges described above.

Priority flow control (PFC). One approach to handling increased buffer occupancy would be to use PFC, a hop-by-hop flow control mechanism.[3] With PFC, if the packets from a particular input port start building up at a congested switch (past a configurable threshold), the switch sends a "pause" frame upstream, stopping that input from sending more traffic until the switch has a chance to drain stored packets. This prevents switch buffers from being overrun. Unfortunately, PFC has a side effect: head-of-line (HoL) blocking [65]. For example, incast traffic to a single server can cause PFC pause frames to be sent one hop upstream towards the source of the traffic. This stops *all* the traffic traversing the paused link, even those flows that are destined to other uncongested egress ports. These flows will be delayed until the packets at the congested port can be drained. Worse, as packets queue up behind a PFC, additional PFC pause frames can be triggered at upstream hops, widening the scope of HoL blocking.

Switch scheduling. Several efforts use switch scheduling to overcome the negative side-effects of elephant flows on the latency of short flows. These proposals range from approximations of fair queuing (e.g., Stochastic Fair Queuing [46], Approximate Fair Queuing [53]) to scheduling policies that prioritize short flows (e.g., pFabric [9], QJump [28], Homa [49]). Our work is orthogonal to the choice of switch scheduling policy, and we present results with priority scheduling and shortest flow first. Scheduling by itself does nothing to reduce buffer occupancy; buffers can fill, causing packet drops or HoL blocking, regardless of scheduling.

Receiver-based congestion control. Because sender-based congestion control schemes generally perform poorly on incast workloads, some researchers have proposed shifting to a scheme where the receiver prevents congestion by explicitly allocating credits to senders for sending traffic. Three examples are NDP [30], pHost [25] and Homa [49]. BFC makes fewer assumptions than these approaches. Homa, for example, assumes knowledge of the flow size distribution and flow length, so that it can assign flows to near-optimal priority queues; this is unavailable with today's TCP socket interface

---

[3]For simplicity, we focus on the case where there is congestion among the traffic at a particular priority level.

and not all applications know flow lengths in advance [13,59]. Homa uses packet spraying to achieve better load balancing, so that congestion primarily occurs at the last hop, where the receiver has complete visibility. However, congestion-free operation of the core is difficult to engineer for widely deployed oversubscribed and asymmetric networks [54,64,66]. Packet spraying can also cause packet reordering, which is incompatible with high-speed end host software and hardware packet handling [35,45]. Other proposals suggest collecting credits generated by a flow's receiver (congestion-controlled by all switches on the flow's path) before sending [22]; at high link speeds, the network state changes rapidly over the feedback delay, making it difficult for the receiver to determine the right rate for credits, similar to sender-based protocols.

2.3 Revisiting Per-hop, Per-Flow Flow Control

Our approach is inspired by work in the early 90s on hop-by-hop credit-based flow control for managing gigabit ATM networks [11,41]. Credit-based flow control was also introduced by multiprocessor hardware designs of the same era [19,38,42]. In these systems, each switch methodically tracks its buffer space, granting permission to send at an upstream switch if and only if there is room in its buffer. In ATM, packets of different flows are buffered in separate queues and are scheduled according to the flows' service requirements. The result is a network that has no congestion loss by design.

An ideal realization of such a per-hop, per-flow flow control scheme has several desirable properties:

(1) Fast reaction: When a flow starts experiencing congestion at a switch, the upstream switch can reduce its rate within a 1-Hop RTT, instead of the end-to-end RTT that it takes for standard congestion control schemes. Likewise, when capacity becomes available at a switch, the upstream switch can increase its rate within a 1-Hop RTT (provided the upstream switch has packets from that flow). Assuming a hardware implementation, the 1-hop RTT consists of the propagation latency and the switch pipeline latency — typically 1-2 μs.[4] This is substantially smaller than the typical end-to-end RTT in data centers (e.g., 10-20 μs), which in addition to multiple switch hops includes the latency at the endpoints.

(2) Buffer pooling: During traffic bursts, a per-hop per-flow flow control mechanism throttles traffic upstream from the bottleneck. This enables the bottleneck switch to tap into the buffers of its upstream neighbors, thereby significantly increasing the ability of the network to absorb bursts.

(3) No HoL blocking: Unlike PFC, there is no HoL blocking or congestion spreading with per-hop per-flow flow control, because switches isolate flows in different queues and perform flow control for each of them separately.

(4) Simple control actions: Flow control decisions in a per-hop per-flow flow control system are simpler to design and

reason about than end-to-end congestion control algorithms because: *(i)* whether to send or pause a flow at a switch depends only on feedback from the immediate next-hop switch (as opposed to multiple potential points of congestion with end-to-end schemes), *(ii)* concerns like fairness are dealt with trivially by scheduling flows at each switch, and therefore flow control can focus exclusively on the simpler task of managing buffer occupancy and ensuring high utilization.

Despite these compelling properties, per-hop per-flow flow control schemes have not been widely deployed, in part because of their high implementation complexity and resource requirements. ATM schemes require per-connection state and large buffers, which are not feasible in today's data center switches. We observe, however, that per-connection switch state is not actually required. Indeed, much of the time, per-connection state is for flows that have no packets queued at the switch, and therefore don't need to be flow controlled.

We define an *active flow* to be a flow with one or more packets queued at the switch. A result of queuing theory is that the number of active flows is surprisingly small for a switch using fair queuing [37,39]. In particular, for an M/G/1-PS (Processor Sharing) queue with Poisson flow arrivals operating at average load $\rho < 1$, the number of active flows has a geometric distribution with mean $\frac{\rho}{1-\rho}$, independent of the link speed or the flow size distribution. Even at load $\rho = 0.9$, the expected number of active flows is only 9. The intuition behind this fact is that a fair queued switch will tend to process short flows quickly, completing them and keeping the number of active flows small.

Data center network workloads are often more bursty than Poisson, leading to longer queues and more active flows. However, the basic principle still holds. Fig. 4 shows the cumulative distribution of the number of active flows for a single bottleneck link operating at different loads and link speeds, using the Google flow size distribution and (bursty) log-normal flow inter-arrival times. The upper graph assumes fair queuing and includes a vertical bar for the number of queues per port on Tofino2. At 100 Gbps, the number of active flows significantly exceeds the number of queues only for loads above 85%, and then only modestly; importantly, the distribution is invariant to link speed, and the trend is for faster links to have more queues. The result holds even more strongly with shortest remaining flow first (SRF) scheduling. By contrast, with FIFO queuing, even a single long flow can cause a large number of small flows to back up behind it, and therefore the number of active flows is much larger.

3 DESIGN

Our goal is to design a practical system for per-hop, per-flow flow control for data center networks. We first describe the constraints on our design (§3.1). We then sketch a plausible strawman proposal that surprisingly turns out to not work well at all (§3.2), and we use that as motivation for our design (§3.3).

---

[4]For example, a 100 m cable has a propagation latency of 500 ns, and a typical data center switch has a pipeline latency around 500 ns [15,20].

(a) Number of active flows vs. link speed, with fair queuing

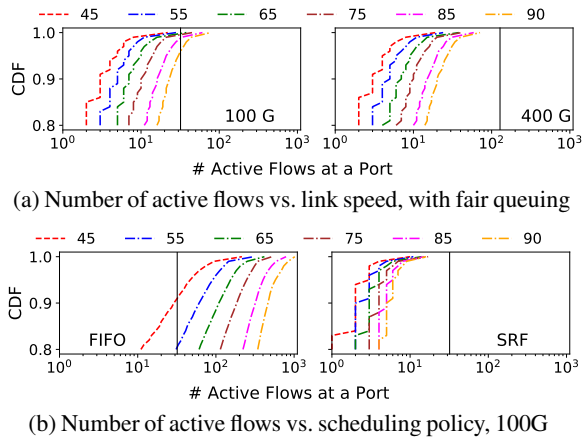(b) Number of active flows vs. scheduling policy, 100G

Figure 4: Number of active flows for different load, link speed, and scheduling policy. Lines correspond to different loads. Flow sizes are from the Google distribution with lognormal ($\sigma = 2$) inter-arrival times.
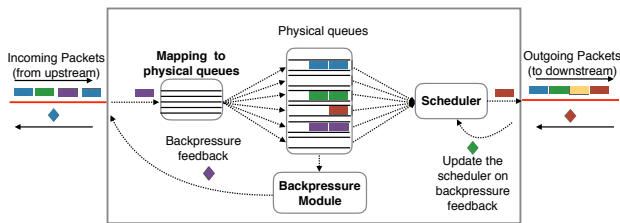

Figure 5: Logical switch components in per-hop, per-flow flow control.

3.1 Design Constraints

Fig. 5 shows the basic components of a per-hop, per-flow flow control scheme (per port). (1) *Mapping to physical queues:* When a packet arrives at the switch, the switch routes the packet to an egress port and maps it to a FIFO queue at that port. This assignment of flows to queues must be consistent, that is, respect packet ordering. (2) *Backpressure module:* Based on queue occupancy, the switch generates backpressure feedback for some flows and sends it upstream. (3) *Scheduler:* The scheduler at each egress port forwards packets from queues while respecting backpressure feedback from the downstream switch.

ATM per-hop per-flow flow control systems [11, 41] roughly followed this architecture, but they would be impractical for modern data centers. First, they assumed per-flow queues and state, but modern switches have a limited number of queues per egress port [17,53] and modest amounts of table memory [18,23]. In particular, it is not possible to maintain switch state for all live connections. Second, earlier schemes did not attempt to minimize buffer occupancy. Instead, they sent backpressure feedback only when the switch was about to run out of buffers. On a buffer-constrained switch, this can result in buffer exhaustion — buffers held by straggler flows can prevent other flows from using those buffers at a later time.

Hardware assumptions. Modern data center switches have made strides towards greater flexibility [12,56], but they are not infinitely malleable and have real resource constraints. We make the following assumptions based on the capabilities of Tofino2.

1. We assume the switch is programmable and supports stateful operations.Tofino2 can maintain millions of register entries, and supports simple constant-time per-packet operations to update the state at line rate [55].
2. The switch has a limited number of FIFO queues per egress port, meaning that flows must be multiplexed onto queues. Tofino2 has 32/128 queues per 100/400G port. The assignment of flows to queues is programmable. The scheduler can use deficit round-robin or priorities among queues, but packets within a queue are forwarded in FIFO order.
3. Each queue can be independently paused and resumed without slowing down forwarding from other queues. When we pause a queue, that pauses *all* of the flows assigned to that queue. The switch can pause/resume each queue directly within the dataplane.

3.2 A Strawman Proposal

We originally thought stochastic fair queuing [46] with per-queue backpressure might meet our goals: use a hash function on the flow header to consistently assign the packets of each flow to a randomly-chosen FIFO queue at its egress port, and pause a queue whenever its buffer exceeds the 1-hop bandwidth-delay product (BDP). For simplicity, use the same hash function at each switch.

This strawman needs only a small amount of state for generating the backpressure feedback and no state for queue assignment. However, with even a modest number of active flows, the birthday paradox implies that there is a significant chance that any specific flow will land in an already-occupied FIFO queue. These collisions hurt latency for two reasons: (1) The packets for the flow will be delayed behind unrelated packets from other flows; for example, a short flow may land behind a long flow. (2) Queue sharing can cause HoL blocking. If a particular flow is paused (because it is congested downstream), all flows sharing the same queue will be delayed. To prevent collisions from affecting tail latency performance, the strawman requires significantly more queues than active flows. For example, at an egress port with n active flows, to achieve fewer than 1% collisions, we would need roughly $100n$ queues.

3.3 Backpressure Flow Control (BFC)

Our design achieves the following properties:

Minimal HoL blocking: We assign flows to queues dynamically. As long as the number of active flows at an egress is less than the number of queues, (with high probability) no two flows share a queue and there is no HoL blocking. When a new flow arrives at the switch, it is assigned to an empty queue if one is available, sharing queues only if all are in use.

Low buffering and high utilization: BFC pauses a flow at the upstream when the queue occupancy exceeds a small threshold. BFC's pause threshold is set aggressively to

reduce buffering. With coarse pausing like PFC, pausing aggressively hurts utilization, but BFC only pauses those flows causing congestion (except when collisions occur). The remaining flows at the upstream can continue transmitting, avoiding under-utilization.

Hardware feasibility: BFC does not require per-flow state, and instead uses an amount of memory proportional to the number of physical queues in the switch. To allow efficient lookup of the state associated with a flow, the state is stored in a flow table, an array indexed using a hash of the flow identifier. The size of this array is set in proportion to the number of physical queues. In our Tofino2 implementation, it consumes less than 10% of the dedicated stateful memory. Critically, the mechanism for generating backpressure and reacting to it is simple and the associated operations can be implemented entirely in the dataplane at line rate.

Generality: BFC does not make assumptions about the network topology or where congestion can occur, and does not require packet spraying like NDP [30] or Homa [49]. Furthermore, it does not assume knowledge of flow sizes or deadlines. Such information can be incorporated into BFC's design to improve small flow performance (see App. A.2), at a cost in deployability.

Idempotent state: Because fiber packets can be corrupted in flight [66], BFC ensures that pause and resume state is maintained idempotently, in a manner resilient to packet loss.

3.3.1 Assigning flows to queues

To minimize sharing of queues and HoL blocking, we dynamically assign flows to empty queues. As long as the flow is active (has packets queued at the switch), subsequent packets for that flow will be placed into the same FIFO queue. Each flow has a unique 5-tuple of the source and destination addresses, port numbers, and protocol; we call this the flow identifier (FID). BFC uses the hash of the FID to track a flow's queue assignment. To simplify locating an empty queue, BFC maintains a bit map of empty queues. When the last packet in a queue is scheduled, BFC resets the corresponding bit for that queue.

With dynamic queue assignment, a flow can be assigned to different queues at different switches. To pause a flow, BFC pauses the queue the flow came from at the upstream switch (called the upstream queue). The pause applies to all flows sharing the same upstream queue with the paused flow. We describe the pause mechanism in detail in §3.3.2. The packet scheduler uses deficit round robin to implement fair queuing among the queues that are not paused.

Since there is a limited number of queues, it is possible that all queues have been allocated when a new flow arrives, at which point HoL blocking is unavoidable. For hardware simplicity, we assign the flow to a random queue in this case. Packets assigned to the same queue are scheduled in FIFO order. The number of active flows is usually small (§2.3), but in certain settings, such as incast, it can exceed the number of queues. BFC's behavior is similar to stochastic fair queuing in such scenarios in that it incurs HoL blocking. BFC still outperforms existing protocols like DCQCN and HPCC except in the most extreme cases (see App. A.1). Even during a large scale incast, BFC can leverage the large number of upstream queues feeding traffic to a bottleneck switch to (1) absorb larger bursts, and (2) limit congestion spreading. In particular, when flows involved in an incast are spread among multiple upstream ports, BFC assigns these flows to separate queues at those ports. As long as the total number of flows does not exceed the total number of queues across *all* of the upstream ports, BFC will not incur HoL blocking at the upstream switches. As the size of the network increases and the fan-in to each switch gets larger, there will be even more queues at the upstream switches to absorb an incast, further reducing congestion spreading.

Mechanism: To keep track of queue assignment, BFC maintains an array indexed by the egress port of a flow and the hash of the FID. All flows that map to the same index are assigned to the same queue. We maintain the following state per entry: the physical queue assignment (`qAssignment`), and the number of packets in the queue from the flows mapped to this entry (`size`). The pseudocode is as follows (we defer switch-specific implementation issues to §6.1):

```
On Enqueue(packet):
    key = <packet.egressPort, hash(packet.FID)>
    if flowTable[key].size == 0:
        reassignQueue = True:
    flowTable[key].size += 1
    if reassignQueue:
        if empty q available at packet.egressPort:
            qAssignment = emptyQ
        else:
            qAssignment = randomQ
        flowTable[key].qAssignment = qAssignment
    packet.qAssignment = flowTable[key].qAssignment

On Dequeue(packet):
    key = <packet.egressPort, hash(packet.FID)>
    flowTable[key].size -= 1
```

In the flow table, if two flows map to the same index they will use the same queue (collision). Since flows going through different egress ports cannot use the same queue, the index also includes the egress port. Index collisions in the flow table can hurt performance. These collisions decrease with the size of the table, but the flow table cannot be arbitrarily large as the switch has a limited stateful memory. In our design, we set the size of the flow table to $100 \times$ the number of queues in the switch. This ensures that if the number of flows at an egress port is less than the number of queues, then the probability of index collisions is less than 1%. If the number of flows exceeds the number of queues, then the index collisions do not matter as there will be collisions in the physical queues regardless. Tofino2 has 4096 queues in aggregate, and hence the size of the flow table is 409,600 entries, which is less than 10% of the switch's dedicated stateful memory.

While using an array is not memory efficient, accessing state involves simple operations. Existing solutions for maintaining flow state either involve slower control plane operations, or are more complex [14, 51]. In the future, if the number of queues increases substantially, we can use these solutions for the flow table; however at the moment, the additional complexity is unnecessary.

3.3.2 Backpressure mechanism

BFC pauses a flow if the occupancy of the queue assigned to that flow exceeds the pause threshold Th. To pause/resume a flow, the switch could signal the flow ID to the upstream switch, which can then pause/resume the queue associated with the flow. While this solution is possible in principle, it is difficult to implement on today's programmable switches. The challenge is that, on receiving a pause, the upstream switch needs to perform a lookup to find the queue assigned to the flow and some additional bookkeeping to deal with cases when a queue has packets from multiple flows (some of which might be paused and some not).

We take a different approach. Switches directly signal to the upstream device to pause/resume a specific queue. Each upstream switch/source NIC inserts its local queue number in a special header field called upstreamQ. The downstream switch uses this information to pause the queue at the upstream.

Mechanism: Recall that, in general, multiple flows can share a queue in rare cases. This has two implications. First, we track the queue length (and not just the flowTable.size) and use that to determine if the flow's upstream queue should be paused. Second, each upstream queue can, in general, have flows sending packets to multiple queues at multiple egresses. We pause an upstream queue if *any* of its flows are assigned a congested queue, and we resume when *none* of its flows have packets at a congested queue (as measured at the time the packet arrived at the switch).

We monitor this using a Pause Counter, an array indexed by the ingress port and the upstreamQ of a packet. The upstream queue is paused if and only if its Pause Counter at the downstream switch is non-zero. On enqueue of a packet, if its flow is assigned a queue that exceeds the pause threshold, we increment the pause counter at that index by 1. When this packet (the one that exceeded Th) leaves the switch we decrement the counter by 1. Regardless of the number of flows assigned to the upstreamQ, it will be resumed only once all of its packets that exceeded the pause threshold (when the packet arrived) have left the switch.

```
On Enqueue(packet):
    key = <packet.ingressPort, packet.upstreamQ>
    if packet.qAssignment.qLength > Th:
        packet.metadata.counterIncr = True
        pauseCounter[key] += 1
        if pauseCounter[key] == 1:
            //Pause the queue at upstream
            sendPause(key)
```

```
On Dequeue(packet):
    key = <packet.ingressPort, packet.upstreamQ>
    if packet.metadata.counterIncr == True:
        pauseCounter[key] -= 1
        if pauseCounter[key] == 0:
            //Resume the queue at upstream
            sendResume(key)
```

To minimize bandwidth consumed in sending pause/resumes, we only send a pause packet when the pause counter for an index goes from 0 to 1, and a resume packet when it goes from 1 to 0. For reliability against pause/resume packets being dropped, we also periodically send a bitmap of the queues that should be paused at the upstream (using the pause counter). Additionally, the switch uses a high priority queue for processing the pause/resume packets. This reduces the number of queues available for dynamic queue assignment by 1, but it eliminates performance degradation due to delayed pause/resume packets.

The memory required for the pause counter is small compared to the flow table. For example, if each upstream switch has 128 queues per egress port, then for a 32-port downstream switch, the pause counter is 4096 entries.

Pause threshold. BFC treats any queue buildup as a sign of congestion. BFC sets the pause threshold Th to 1-Hop BDP at the queue drain rate. Let N_{active} be the number of *active queues* at an egress, i.e. queues with data to transmit that are not paused, $HRTT$ be the 1-Hop RTT to the upstream, and μ be the port capacity. Assuming fair queuing as the scheduling policy, the average drain rate for a queue at the egress is μ/N_{active}. The pause threshold Th is thus given by $(HRTT) \cdot (\mu/N_{active})$. When the number of active queues increases, Th decreases. In asymmetric topologies, egress ports can have different link speeds; as a result, we calculate a different pause threshold for every egress based on its speed. Similarly, ingress ports can have different 1-Hop RTTs. Since a queue can have packets from different ingresses, we use the max of $HRTT$ across all the ingresses to calculate Th. We use a pre-configured match-action table indexed with N_{active} and μ to compute Th.

BFC does not guarantee that a flow will never run out of packets due to pausing. First, a flow can be paused unnecessarily if it is sharing its upstream queue with other paused flows. Second, a switch only resumes an upstream queue once all its packets (that exceeded the pause threshold when they arrived) have left the downstream switch. Since the resume takes an $HRTT$ to take effect, a flow can run out of packets at the downstream switch for an $HRTT$, potentially hurting utilization. However, this scenario is unlikely — a pause only occurs when a queue builds up, typically because multiple flows are competing for the same egress port. In this case, the other flows at the egress will have packets to occupy the link, preventing under-utilization.

We might reduce the (small) chance of under-utilization by resuming the upstream queue earlier, for example, when a flow's queue at the downstream drops below Th, or more precisely, when *every* queue (with a flow from the same

upstream queue) drops below Th. Achieving this would require extra bookkeeping, complicating the design.

Increasing the pause threshold would reduce the number of pause/resumes generated, but only at the expense of increased buffering (Fig. 7). In App. C, we analyze the impact of Th on under-utilization and peak buffer occupancy in a simple model, and we show that a flow runs out of packets at most 20% of the time when Th is set to 1-hop BDP. Our evaluation results show that BFC achieves much better throughput than this worst case in practice (Table 1, §6).

Sticky queue assignment: Using `upstreamQ` for pausing flows poses a challenge. Since a switch does not know the current queue assignment of a flow at the upstream, it uses the `upstreamQ` conveyed by the last packet of the flow to pause a queue. However, if a flow runs out of packets at the upstream switch (e.g., because it was bottlenecked at the downstream switch but not the upstream), then its queue assignment may change for subsequent packets, causing it to temporarily evade the pause signal sent by the downstream switch. Such a flow will be paused again when the downstream receives packets with the new `upstreamQ`. The old queue will likewise be unpaused when its last packet (that exceeded Th) departs the downstream switch.

To reduce the impact of such queue assignment changes, we add a timestamp to the flow table state, updated whenever a packet is enqueued or dequeued. A new queue assignment only happens if the `size` value in the flow table is 0, and the timestamp is older than a "sticky threshold" (i.e., the entry in the flow table has had no packets in the switch for at least this threshold). Since with BFC's backpressure mechanism a flow can run out of packets for an $HRTT$, we set the sticky threshold to a small multiple of $HRTT$ (2 $HRTT$).

While sticky queue assignments reduce the chance that a backlogged flow will change queues, it doesn't completely eliminate it (e.g., packets from the same flow may arrive slower than this interval due to an earlier bottleneck). Such situations are rare, and we found that BFC performs nearly identically to an ideal (but impractical) variant that pauses flows directly using the flow ID without sticky queue assignments.

4   TOFINO2 IMPLEMENTATION

We implemented BFC in Tofino2, a to-be-released P4-based programmable switch ASIC with a Reconfigurable Match Table (RMT) architecture [17]. A packet in Tofino2 first traverses the ingress pipeline, followed by the traffic manager (TM) and finally the egress pipeline. Tofino2 has four ingress and four egress RMT pipelines. Each pipeline has multiple stages, each capable of doing stateful packet operations. Ingress/egress ports are statically assigned to pipelines.

Bookkeeping: The flow table and pause counter are both maintained in the ingress pipeline. The flow table contains three values for each entry and is thus implemented as three separate register arrays (one for each value), updated one after the other.

Multiple pipelines: The flow table is *split* across the four ingress pipelines, and the size of the table in each ingress pipeline is 25 $\times$ the number of queues. During normal operation, packets of an active flow arrive at a single ingress pipeline (same ingress port). Since the state for a flow only needs to be accessed in a single pipeline, we can split the flow table. However, splitting can marginally increase collisions if the incoming flows are distributed unevenly among the ingress pipelines. Similarly, the pause counter is split among the ingress pipelines. An ingress pipeline contains the pause counter entries corresponding to its own ingress ports.

Gathering queue depth information: We need queue depth information in the ingress pipeline for pausing and dynamic queue assignment. Tofino2 has an inbuilt feature tailored for this task. The TM can communicate the queue depth information for all the queues in the switch to all the ingress pipelines without consuming any additional ingress cycles or bandwidth. The bitmap of empty queues is periodically updated with this data, with a different rotating starting point per pipeline to avoid new flows from being assigned to the same empty queue.

Communicating from egress to ingress pipeline: The enqueue operations described earlier are executed in the ingress pipeline when a packet arrives. Dequeue operations should happen at the egress but the bookkeeping data structures are at the ingress. To solve this, in the egress pipeline, we mirror packets as they exit and recirculate the header of the mirrored packet back to the ingress pipeline it came from. The dequeue operations are executed on the recirculated packet header.

Recirculating packets involves two constraints. First, the switch has dedicated internal links for recirculation, but the recirculation bandwidth is limited to 12% of the entire switch capacity. Second, the recirculated packet consumes an additional ingress cycle. The switch has a cap on the number of packets it can process every second (pps capacity).

Most workloads have an average packet size greater than 500 bytes [16], and Tofino2 is designed with enough spare capacity in bandwidth and pps to handle header recirculation for every packet for those workloads (with room to spare). If the average packet size is much smaller, we can reduce recirculations by sampling packets for recirculation (described in App. A.8).

Recirculation is not fundamental to BFC. For example, Tofino2 has native support for PFC bookkeeping in the TM. Likewise, if BFC bookkeeping was implemented in the TM, it would not need recirculation. Similarly, in switches with a disaggregated RMT architecture [23] where the same memory can be accessed at both the ingress and egress, there is no need for recirculation.

5 DISCUSSION

Guaranteed losslessness. BFC does not guarantee lossless-ness. In particular, a switch in BFC pauses an `upstreamQ` only after receiving a packet from it. This implies an `upstreamQ` can send packets for up to an *HRTT* to the bottleneck switch before being paused, even if the switch is congested. In certain mass incast scenarios, this might be sufficient to trigger drops. Using credits [11, 41] could address this at the cost of added complexity. We leave an investigation of such prospective variants of BFC to future work. In our evaluation with realistic switch buffer sizes, BFC never incurred drops except under a 2000-to-1 incast (§6.3) and even then only 0.007% of the packets were dropped.

Deadlocks: Pushback mechanisms like PFC have been shown to be vulnerable to deadlocks in the presence of cyclic buffer dependencies (CBD) or misbehaving NICs [29, 31]. BFC NICs do not generate any backpressure and as a result cannot cause deadlocks. Since NICs always drain, in the absence of CBD, BFC cannot have deadlocks (see App. B for a formal proof). A downstream switch in BFC *will* resume an `upstreamQ` if it drains all the packets sent by the `upstreamQ`. If a downstream is not deadlocked, it will eventually drain packets from the upstream, and as a result, the corresponding upstream cannot be deadlocked.

To prevent CBD, we can reuse prior approaches for deadlock prevention. These approaches can be classified into two categories. The first is to redesign routing protocols to avoid installing routes that might cause CBD [57, 58]. The other is to identify a subset of possible ingress/egress pairs that are provably CBD free, and only send pause/resume along those pairs [32]. For a fat-tree topology, this would allow up-down paths but not temporary loops or detour routes [44]. In BFC, we use the latter approach. Given a topology, we pre-compute a match action table indexed by the ingress and egress port, and simply elide the backpressure pause/resume signal if it is disallowed. See App. B for details.

Incremental Deployment: In a full deployment, BFC would not require end-to-end congestion control. In a partial deployment, we advocate some form of end-to-end congestion control, such as capping the number of inflight packets of a flow. A common upgrade strategy is to upgrade switches more rapidly than server NICs. If only switches and not NICs are running BFC, capping inflight packets prevents a source NIC from overrunning the buffers of the first hop switch. The same strategy can be used for upgrading one cluster's switches before the rest of the data center [64]. In our evaluation, we show incremental deployment would have some impact on buffer occupancy at the edge but minimal impact on performance (App. A.8).
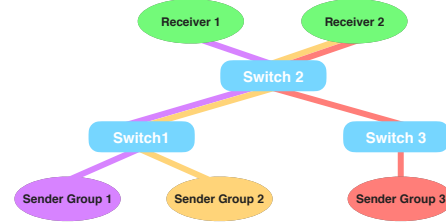
Figure 6: **Testbed topology.** The colored lines show the path for different flow groups.

(a) Queue Length   (b) Under-utilization
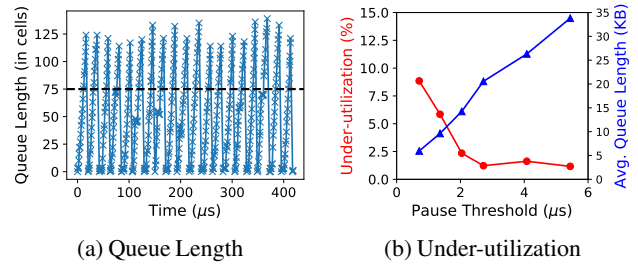
Figure 7: **Queue length and under-utilization.** 2 flows are competing at a 100 Gbps link. Cell size is 176 bytes. BFC achieves high utilization and low buffering.

6 EVALUATION

We present a proof-of-concept evaluation of our Tofino2 implementation. To compare performance of BFC against existing schemes, we perform large scale ns-3 [4] simulations.

6.1 Tofino2 evaluation

Testbed: For evaluation, we were able to gain remote access to a Tofino2 switch. Using a single switch, we created a simple multi-switch topology (Fig. 6) by looping back packets from the egress port back into the switch. All the ports are 100 Gbps, each port has 16 queues.[5] The experiments include three groups of flows.

- Sender Group 1 → Switch 1 → Switch 2 → Receiver 1.
- Sender Group 2 → Switch 1 → Switch 2 → Receiver 2.
- Sender Group 3 → Switch 3 → Switch 2 → Receiver 2.

To generate traffic we use the on-chip packet generator with no end-to-end congestion control.

Low buffering, high utilization: Fig. 7a shows the queue length for a flow when two flows are competing at a link (a group 2 flow is competing with a group 3 flow at the switch 2 → receiver 2 link). The pause threshold is shown as a horizontal black line. BFC's pausing mechanism is able to limit the queue length near the pause threshold (*Th*). The overshoot from *Th* is for two reasons. First, it takes an *HRTT* for the pause to take effect. Second, Tofino2 has small hardware queues after the egress pipeline, and a pause from the downstream cannot pause packets already in these hardware queues.

Notice that the queue length goes to 0 temporarily. Recall that a downstream switch only resumes the `upstreamQ`

---

[5]For 100 Gbps ports, Tofino2 has 32 queues, but in loopback mode only 16 queues are available.
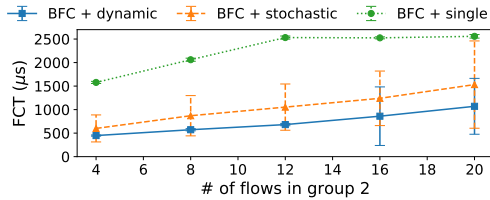
Figure 8: **Congestion spreading.** Dynamic queue assignment reduces HoL blocking, improving FCTs on average and at the tail.

when it has drained all the packets from the `upstreamQ` that exceeded Th. As a result, a flow at the downstream can run out of packets for an $HRTT$. This can cause under-utilization when the queues for the two flows go empty simultaneously. We repeat the above experiment but vary the pause threshold. Fig. 7b shows the average queue length and the under-utilization of the congested link. With a pause threshold of 2 μs, BFC achieves close to 100% utilization with an average queue length of 15 KB.

Queue assignment and congestion spreading: We next evaluate the impact of queue assignment on HoL blocking and performance. We evaluate three different queue assignment strategies with BFC's backpressure mechanism: (1) "BFC + single": All flows are assigned to a single queue (similar to PFC); (2) "BFC + stochastic": Flows are assigned to queues using stochastic hashing; (3) "BFC + dynamic": Dynamic queue assignment as described in §3.3.1.

The setup consists of two group 1 flows, eight group 3 flows, and a number of group 2 flows varied between four to twenty. All flows are 1.5 MB in size. The experiment is designed such that for group 2 and 3 flows, the bottleneck is the switch 2 → receiver 2 link. The bottleneck for group 1 flows is the switch 1 → switch 2 link. Switch 2 will pause queues at switch 1 in response to congestion from group 2 flows. Notice that group 1 and group 2 flows are sharing the switch 1 → switch 2 link. If a group 1 flow shares a queue with a group 2 flow (a collision), the backpressure due to the group 2 flow can slow down the group 1 flow, causing HoL blocking and increasing its flow completion time (FCT) unnecessarily.

Fig. 8 shows the average FCT for group 1 flows across four runs. The whiskers correspond to one standard deviation in the FCT. BFC + single achieves the worst FCT as group 1 and 2 flows always share a queue. With stochastic assignment, the FCT is substantially lower, but the standard deviation in FCT is high. In some runs, group 1 and 2 flows don't share a queue and there is no HoL blocking. In other runs, due to the stochastic nature of assignment, they do share a queue (even when there are other empty queues), resulting in worse performance. With dynamic assignment, BFC achieves the lowest average FCT and the best tail performance. In particular, the standard deviation is close to 0 when the number of flows at the switch 1 → switch 2 link (group 1 + group 2 flows) is lower than the number of queues. In such scenarios, group 1 flows consistently incur no collisions. When the number of flows exceed the queues, collisions are

inevitable, and the standard deviation in FCT increases.

6.2 Simulation-based evaluation

We also implemented BFC in ns-3 [4]. For DCQCN we use [5], for ExpressPass we use [1], and for all other schemes we use [3].

6.2.1 Setup

Network Topology: We use a Clos topology with 128 leaf servers, 8 top of the rack (ToR) switches and 8 Spine switches (2:1 over subscription). Each Spine switch is connected to all the ToR switches, each ToR has 16 servers, and each server is connected to a single ToR. All links are 100 Gbps with a propagation delay of 1 us. The maximum end-to-end base round trip time (RTT) is 8 μs and the 1-Hop RTT is 2 μs. The switch buffer size is set to 12 MB. Relative to the ToR switch capacity of 2.4 Tbps, the ratio of buffer size to switch capacity is 40 μs, the same as Broadcom's Tomahawk3 from Fig. 1. We use an MTU of 1 KB. Unless specified otherwise, we use Go-Back-N for retransmission, flow-level ECMP for load balancing, and the standard shared buffer memory model implemented in existing switches [20].

Comparisons: *HPCC:* HPCC uses explicit link utilization information from the switches to reduce buffer occupancy and drops/PFCs at the congested switch. We use the parameters from the paper, $\eta = 0.95$ and $maxStage = 5$. The dynamic PFC threshold is set to trigger when traffic from an input port occupies more than 11% of the free buffer (as in the HPCC paper). We use the same PFC thresholds for DCQCN and DCTCP. *HPCC-PFC:* This version replaces PFC with perfect retransmission. On a packet drop, the switch informs the sender directly, which then retransmits the dropped packet. We choose this (potentially impractical) strategy to provide a bound on the performance that can be achieved using any retransmission scheme. *DCQCN:* DCQCN uses ECN bits and end-to-end control to manage buffer use at the congested switch. The ECN threshold triggers before PFC (K_{min} = 100KB and K_{max} = 400KB). *DCTCP:* The ECN threshold is same as DCQCN. Flows start at line rate to avoid degradation in FCTs from slow-start. *ExpressPass:* In ExpressPass, senders transmit data based on credits generated by the receiver. These credits are rate-limited at the switches to avoid congestion. We chose $\alpha = 0.5, w_{init} = 0.0625$ and a credit buffer size of 16 credits. The ExpressPass simulator does not follow a shared buffer model; instead it assumes dedicated per-port buffers. To eliminate drops, we supplied a high per-port buffer value of 75 MB. There is no PFC. *BFC:* We use 32 physical queues per port (consistent with Tofino2) and our flow table has 76K entries. The flow table takes 400 KB of memory. We chose per-flow fair queuing as our scheduling mechanism; all the comparison schemes strive for per-flow fairness, thus, fair queuing provides for a just comparison.

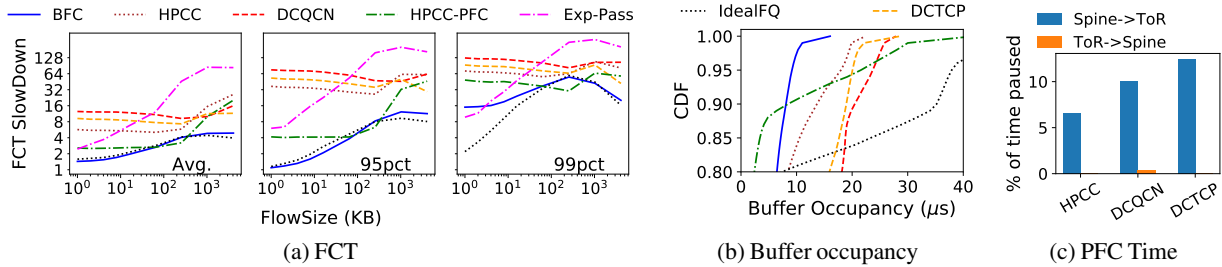(a) FCT   (b) Buffer occupancy   (c) PFC Time

Figure 9: Google distribution with 55% load + 5% 100-1 incast. BFC tracks the ideal behavior, improves FCTs, and reduces buffer occupancy. For FCT slowdown, both the x and y axis are log scaled.

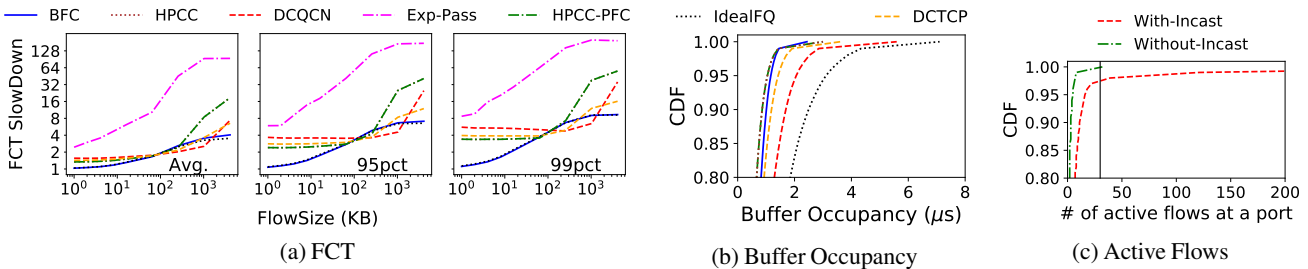

(a) FCT   (b) Buffer Occupancy   (c) Active Flows

Figure 10: FCT slowdown and buffer occupancy for Google distribution with 60% load. For all the schemes, PFC was never triggered. Part (c) shows the CDF of active flows at a port with and without incast, with the vertical bar showing the total number of queues per port.

Ideal-FQ: To understand how close BFC comes to optimal performance, we simulate ideal fair queuing with infinite buffering at each switch. The NICs cap the in-flight packets of a flow to 1 BDP. Note that infinite buffering is not realizable in practice; its role is to bound how well we could possibly do.

Sensitivity to parameters: All systems were configured to achieve full throughput for a single flow on an unloaded network. For end-to-end schemes, the choice of parameters governs the trade-off between the performance of short flows (through reduced queuing) and long flows (higher link utilization). We perform parameter sensitivity analysis for HPCC, DCTCP and ExpressPass in App. A.4.

Performance metrics: We consider three performance metrics: (1) FCT normalized to the best possible FCT for the same size flow, running at link rate (referred as the FCT slowdown); (2) Overall buffer occupancy at the switch; (3) Throughput of individual flows.

Workloads: We synthesized a trace to match the flow size distributions from the industry workloads discussed in Fig. 2: (1) Aggregated workload from all applications in a Google data center; (2) a Hadoop cluster at Facebook (FB_Hadoop). The flow arrival pattern is open-loop and follows a *bursty* log-normal inter-arrival time distribution with σ = 2.[6] For each flow arrival, the source-destination pair is derived from a uniform distribution. We consider scenarios with and without incast, different traffic load settings, and incast ratios. Since our topology is oversubscribed, on average links in the core (Spine-ToR) will be more congested than the ToR-leaf server links. In our experiments, by X% load we mean X% load on

the links in the core.

6.2.2 Performance

Fig. 9 and 10 show our principal results. The flow sizes are drawn from the Google distribution and the average load is set to 60% of the network capacity. For Fig. 9 (but not Fig. 10), 5% of the traffic (on average) is from incast flows. The incast degree is 100-to-1 and the size is 20 MB in aggregate. A new incast event starts every 500 μs. Since the best-case completion time for an incast is 1.6 ms (20 MB/100 Gbps), multiple incasts coexist simultaneously in the network. We report the FCT slowdowns at the average, 95th and 99th percentile, the tail buffer occupancy (except for ExpressPass simulations which do not follow the shared buffer model), and the fraction of time links were paused due to PFC. We report the FCT slowdowns for the incast traffic separately in App. A.12.

Out of all the schemes, DCQCN is worst on latency for small flow sizes, both at the average and the tail. Compared to DCQCN, DCTCP improves latency as it uses per-ACK feedback instead of periodic feedback via QCN. However, the frequent feedback is not enough, and the performance is far from optimal (Ideal-FQ). The problem is that both DCQCN and DCTCP are slow in responding to congestion. Since flows start at line rate, a flow can build up an entire end-to-end bandwidth-delay product (BDP) of buffering (100 KB) at the bottleneck before there is any possibility of reducing its rate. The problem is aggravated during incast events. The bottleneck switch can potentially accumulate one BDP of packets per incast flow (10 MB in aggregate for 100-to-1 incast).

Both protocols have low throughput for long flows. When capacity becomes available, a long flow may fail to ramp up quickly enough, reducing throughput and shifting its work

---

[6]Most prior work evaluates using Poisson flow arrivals [22, 49], but we use the more bursty Lognormal as it provides a more challenging case for BFC.

to busier periods where it can impact other flows. Moreover, on sudden onset of congestion, a flow may not reduce its rate fast enough, slowing short flows.

HPCC improves on DCQCN and DCTCP by using link utilization instead of ECN and a better control algorithm. Compared to DCQCN and DCTCP, HPCC reduces tail latency, tail buffer occupancy, and PFC pauses (in case of incast). Compared to BFC, however, HPCC has 5-30× worse tail latency for short flows with incast, and 2.3-3× worse without. Long flows do worse with HPCC than DCQCN and DCTCP since HPCC deliberately targets 95% utilization and very small queues to improve tail latency for short flows.

With ideal retransmission, HPCC performance improves, especially for short and medium flows. However, HPCC without PFC has higher tail buffer occupancy and suffers packet loss. Compared to BFC, overall performance is still worse for both long and short flows.

Across all systems, ExpressPass achieves the worst throughput for long flows. In ExpressPass, the receiver can generate unnecessary credits for an additional RTT before learning that a flow is finished. These credits are considered "wasted" as the sender cannot transmit packets in response, and can therefore cause link under-utilization. Credit waste and the corresponding under-utilization increase with faster link speeds and/or when the flow sizes get shorter (see §6.3 and §7 in [22]).

Ideal-FQ achieves lower latency than all the schemes, but its buffer occupancy can grow to an unfeasible level.

BFC achieves the best FCTs (both average and tail) among all the schemes. Without incast, BFC performance closely tracks optimal. With incast, incoming flows exhaust the number of physical queues, triggering HoL blocking and hurting tail latency. This effect is largest for the smallest flows at the tail. Fig. 10c shows the CDF of the number of active flows at a port. In the absence of incast, the number of active flows is smaller than the total queues 99% of the time, and collisions are rare. With incast, the number of active flows increases, causing collisions. However, the tail latency for short flows with BFC is still 5-30× better than existing schemes. BFC also improves the performance of incast flows, achieving 2× better FCTs at the tail compared to HPCC (see App. A.12).

Note that, compared to BFC and Ideal-FQ, latency for medium flows (200-1000KB) is slightly better with existing schemes. Because they slow down long flows relative to perfect fairness, medium flows have room to get through more quickly. Conversely, tail slowdown is better for long flows than medium flows with BFC and Ideal-FQ. Long flows achieve close to the long term average available bandwidth, while medium flows are more affected by transient congestion.

Another workload: We repeated the experiment in Fig. 9 and Fig. 10 with the Facebook distribution. Fig. 11 shows the 99th percentile FCT slowdown. The trends in the FCT slowdowns are similar to that of the Google distribution, except that ExpressPass performs better since it incurs fewer wasted

(a) 55% + 5% 100-1 incast   (b) 60%
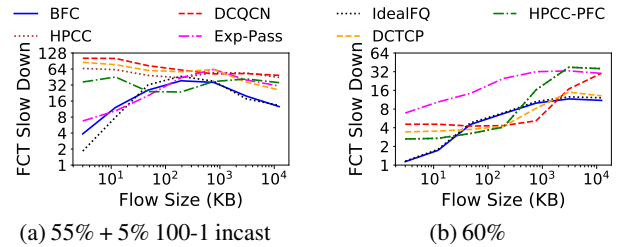
Figure 11: FCT slowdown (99th percentile) for Facebook distribution with and without incast.

(a) Average FCT for long flows   (b) Tail FCT for short flows
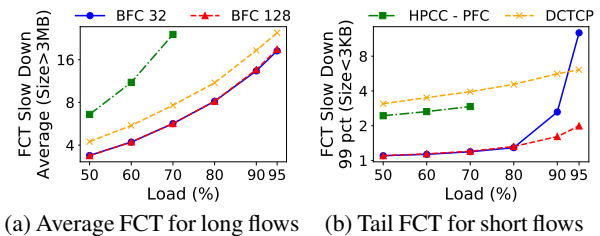
Figure 12: Average FCT slowdown for long flows, and 99th percentile tail FCT slowdown for small flows, as a function of load.

credits (as a percentage) for the Facebook workload, which has larger flows. We omit other statistics presented earlier in the interest of space, but the trends are similar to Fig. 9 and 10. Henceforth, all the experiments use the Facebook workload.

6.3 Stress-testing BFC

In this section we stress-test BFC under high load and large incast degree. Flow arrivals follow a bursty log-normal distribution ($\sigma = 2$). We evaluate BFC under two different queue configurations: (1) 32 queues per port (BFC 32); (2) 128 queues per port (BFC 128). We show the average slowdown for long flows (> 3MB) and 99th percentile slowdown for short flows (< 3KB).

Load: Fig. 12 shows the performance as we vary the average load from 50 to 95% (without incast). HPCC only supports loads up to 70%. At higher loads, it becomes unstable (the number of outstanding flows grows without bound), in part due to the overhead of the INT header (80 B per-packet). All other schemes were stable across all load values.

At loads ≤ 80%, BFC 32 achieves both lower tail latency (Fig. 12b) for short flows and higher throughput for long flows (Fig. 12a). The tail latency for short flows is close to the perfect value of 1. At higher loads, flows remain queued at the bottleneck switch for longer periods of time, raising the likelihood that we run out of physical queues, leading to head of line blocking. This particularly hurts tail performance for short flows as they might be delayed for an extended period if they are assigned to the same queue as a long flow. At the very high load of 95%, the HoL blocking degrades tail latency substantially for BFC 32. However, it still achieves good link utilization, and the impact of collisions is limited for long flows.

Increasing the number of queues reduces collisions and the associated HoL blocking. BFC 128 achieves better tail latency for short flows at load ≥ 90%.
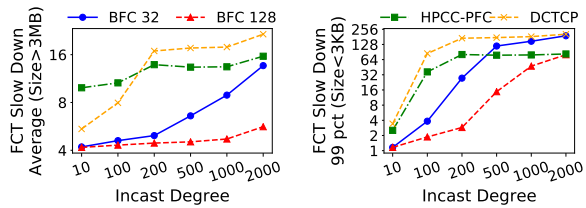
(a) Average FCT for long flows   (b) Tail FCT for short flows

Figure 13: Average FCT slowdown for long flows, and 99th percentile tail FCT slowdown for small flows, as a function of incast degree.
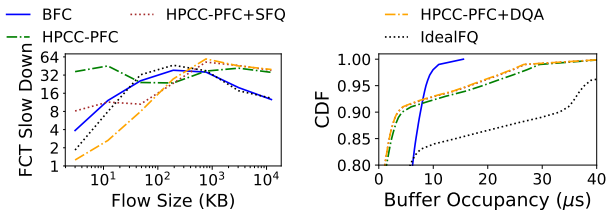
(a) FCT SlowDown   (b) Buffer Occupancy

Figure 14: FCT slowdown (99th percentile) and buffer occupancy of HPCC variants, using the setup in Fig. 11a.

Incast degree: If the size of an incast is large enough, it can exhaust physical queues and hurt performance. Fig. 13 shows the effect of varying the degree of incast on performance. The average load is 60% and includes a 5% incast. The incast size is 20 MB in aggregate, but we vary the degree of incast from 10 to 2000.

For throughput, both BFC 32 and BFC 128 perform well as long as the incast degree is moderate compared to the number of queues. Both start to degrade when the incast degree exceeds $8\times$ the number of queues per port. Till this point, BFC can leverage the FanIn from the larger number of upstream queues (and greater aggregate upstream buffer space) to keep the incast from impeding unrelated traffic. As the incast degree scales up further, BFC 32 is able to retain some of its advantage relative to HPCC and DCTCP.

For high incast degree, the tail latency for short flows becomes worse than HPCC. The tail is skewed by the few percent of small requests that happen to go to the same destination as the incast. (Across the 128 leaf servers in our setup, several servers are the target of an incast at any one time, and these also receive their share of normal traffic.) As the incast degree increases, more small flows share physical queues with incast flows, leading to more HoL blocking.

In App. A.1, we further explore this issue with microbenchmarks designed to trigger a variable number of active flows at the bottleneck switch. We show that by adding a very simple end-to-end control mechanism to BFC, we can ameliorate the impact of HoL blocking while still fully utilizing the link.

6.4 Dynamic Queue Assignment

We next consider the effect of applying BFC's dynamic queue assignment separately from the backpressure mechanism. For this, we modified HPCC with idealized re-

transmission (HPCC-PFC) to add stochastic fair queuing (HPCC-PFC+SFQ) and dynamic queue assignment (HPCC-PFC+DQA). To match BFC, we use 32 physical queues with HPCC. We repeat the experiment from Fig. 11a, showing tail slowdown and buffer occupancy for the HPCC variants, BFC, and IdealFQ in Fig. 14.

Adding SFQ to HPCC improves short flow latency by isolating them from long flows in different queues, but it still suffers from more collisions (and thus higher tail latency for short flows) than DQA. DQA on its own, however, has no benefit for long flows: since HPCC is unable to adapt to rapid changes in the number of flows (and the fair-share rate), it is unable to fully utilize the link for long flows, even with DQA. Moreover, both HPCC-PFC+SFQ and HPCC-PFC+DQA build deep buffers and experience drops at the same rate as HPCC-PFC. Notice that HPCC's lower throughput for long flows favors short flows to such an extent that HPCC-PFC+DQA achieves better tail latency for short flows than both BFC and IdealFQ.

6.5 Additional Experiments

In App. A, we use our simulation framework to further characterize the limits of BFC, compare BFC to Homa, as well as study the impact of priority scheduling, parameter selection, locality in the traffic matrix, slow start, incast labelling, and other factors.

7 CONCLUSION

In this paper, we present Backpressure Flow Control (BFC), a practical congestion control architecture for data center networks. BFC provides per-hop per-flow flow control, but with bounded state, constant-time switch operations, and careful use of buffers. Switches dynamically assign flows to physical queues, allowing fair scheduling among competing flows and use selective backpressure to reduce buffering with minimal head of line blocking. Relative to existing end-to-end congestion control schemes, BFC improves short flow tail latency and long flow utilization for networks with high bandwidth links and bursty traffic. We demonstrate BFC's feasibility by implementing it on Tofino2, a state-of-art P4-based programmable hardware switch. In simulation, compared to several deployed end-to-end schemes, BFC achieves $2.3\text{-}60\times$ lower tail latency for short flows and $1.6\text{-}5\times$ better average completion time for long flows.

REFERENCES

[1] Express pass simulation. https://github.com/kaist-ina/ns2-xpass.

[2] Homa simulation. https://github.com/PlatformLab/HomaSimulation/tree/omnet_simulations/RpcTransportDesign.

[3] Hpcc simulation. https://github.com/alibaba-edu/High-Precision-Congestion-Control.

[4] Network simulator 3. https://www.nsnam.org.

[5] Ns-3 simulator for rdma. https://github.com/bobzhuyb/ns3-rdma.

[6] Atul Adya, Robert Grandl, Daniel Myers, and Henry Qin. Fast key-value stores: An idea whose time has come and gone. In *Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS 2019, Bertinoro, Italy, May 13-15, 2019*, pages 113–119. ACM, 2019.

[7] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. CONGA: distributed congestion-aware load balancing for datacenters. In Fabián E. Bustamante, Y. Charlie Hu, Arvind Krishnamurthy, and Sylvia Ratnasamy, editors, *ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014*, pages 503–514. ACM, 2014.

[8] Mohammad Alizadeh, Albert G. Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). In Shivkumar Kalyanaraman, Venkata N. Padmanabhan, K. K. Ramakrishnan, Rajeev Shorey, and Geoffrey M. Voelker, editors, *Proceedings of the ACM SIGCOMM 2010 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, New Delhi, India, August 30 -September 3, 2010*, pages 63–74. ACM, 2010.

[9] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: minimal near-optimal datacenter transport. In Dah Ming Chiu, Jia Wang, Paul Barford, and Srinivasan Seshan, editors, *ACM SIGCOMM 2013 Conference, SIGCOMM'13, Hong Kong, China, August 12-16, 2013*, pages 435–446. ACM, 2013.

[10] Amazon. Amazon Web Services. https://aws.amazon.com/s3/.

[11] Thomas E. Anderson, Susan S. Owicki, James B. Saxe, and Charles P. Thacker. High speed switch scheduling for local area networks. *ACM Trans. Comput. Syst.*, 11(4):319–352, 1993.

[12] Arista. Arista 7170 Multi-function Programmable Networking. https://www.arista.com/assets/data/pdf/Whitepapers/7170_White_Paper.pdf.

[13] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Information-agnostic flow scheduling for commodity data centers. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 455–468, 2015.

[14] Tom Barbette, Chen Tang, Haoran Yao, Dejan Kostic, Gerald Q. Maguire Jr., Panagiotis Papadimitratos, and Marco Chiesa. A high-speed load-balancer design with guaranteed per-connection-consistency. In Ranjita Bhagwan and George Porter, editors, *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 667–683. USENIX Association, 2020.

[15] Barefoot. Tofino: World's Fastest P4-Compatible Ethernet Switch ASICs. https://www.barefootnetworks.com/products/brief-tofino/.

[16] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Understanding data center traffic characteristics. *Comput. Commun. Rev.*, 40(1):92–99, 2010.

[17] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: programming protocol-independent packet processors. *Comput. Commun. Rev.*, 44(3):87–95, 2014.

[18] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *Proceedings of the ACM SIGCOMM 2013 Conference*, SIGCOMM '13, page 99–110, New York, NY, USA, 2013. Association for Computing Machinery.

[19] Eric A. Brewer and Bradley C. Kuszmaul. How to get good performance from the CM-5 data network. In Howard Jay Siegel, editor, *Proceedings of the 8th International Symposium on Parallel Processing, Cancún, Mexico, April 1994*, pages 858–867. IEEE Computer Society, 1994.

[20] Broadcom. StrataXGS. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs.

[21] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-Based Congestion Control. *ACM Queue*, 14(5):50:20–50:53, October 2016.

[22] Inho Cho, Keon Jang, and Dongsu Han. Credit-scheduled delay-bounded congestion control for datacenters. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*, pages 239–252. ACM, 2017.

[23] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, Ariel Orda, and Tom Edsall. drmt: Disaggregated programmable switching. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*, pages 1–14. ACM, 2017.

[24] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, 2013.

[25] Peter Xiang Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. phost: distributed near-optimal datacenter transport over commodity network fabric. In Felipe Huici and Giuseppe Bianchi, editors, *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies, CoNEXT 2015, Heidelberg, Germany, December 1-4, 2015*, pages 1:1–1:12. ACM, 2015.

[26] Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, and Mohammad Alizadeh. Juggler: a practical reordering resilient network stack for datacenters. In Cristian Cadar, Peter R. Pietzuch, Kimberly Keeton, and Rodrigo Rodrigues, editors, *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*, pages 20:1–20:16. ACM, 2016.

[27] Google. Google Cloud Platform. https://cloud.google.com.

[28] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. Queues don't matter when you can JUMP them! In *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015*, pages 1–14. USENIX Association, 2015.

[29] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over commodity ethernet at scale. In Marinho P. Barcellos, Jon Crowcroft, Amin Vahdat, and Sachin Katti, editors, *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, pages 202–215. ACM, 2016.

[30] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*, pages 29–42. ACM, 2017.

[31] Shuihai Hu, Yibo Zhu, Peng Cheng, Chuanxiong Guo, Kun Tan, Jitendra Padhye, and Kai Chen. Deadlocks in datacenter networks: Why do they form, and how to avoid them. In Bryan Ford, Alex C. Snoeren, and Ellen W. Zegura, editors, *Proceedings of the 15th ACM Workshop on Hot Topics in Networks, HotNets 2016, Atlanta, GA, USA, November 9-10, 2016*, pages 92–98. ACM, 2016.

[32] Shuihai Hu, Yibo Zhu, Peng Cheng, Chuanxiong Guo, Kun Tan, Jitendra Padhye, and Kai Chen. Tagger: Practical PFC deadlock prevention in data center networks. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies, CoNEXT 2017, Incheon, Republic of Korea, December 12 - 15, 2017*, pages 451–463. ACM, 2017.

[33] Intel. Tofino2. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html. 2020.

[34] Lavanya Jose, Stephen Ibanez, Mohammad Alizadeh, and Nick McKeown. A distributed algorithm to calculate max-min fair rates without per-flow state. *Proc. ACM Meas. Anal. Comput. Syst.*, 3(2):21:1–21:42, 2019.

[35] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas E. Anderson. TAS: TCP acceleration as an OS service. In George Candea, Robbert van Renesse, and Christof Fetzer, editors, *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, pages 24:1–24:16. ACM, 2019.

[36] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. In-band network telemetry via programmable dataplanes. 2015.

[37] Leonard Kleinrock. *Queueing systems, volume 2: Computer applications*, volume 66. wiley New York, 1976.

[38] Smaragda Konstantinidou and Lawrence Snyder. Chaos router: Architecture and performance. In Zvonko G. Vranesic, editor, *Proceedings of the 18th Annual International Symposium on Computer Architecture. Toronto, Canada, May, 27-30 1991*, pages 212–221. ACM, 1991.

[39] Abdesselem Kortebi, Luca Muscariello, Sara Oueslati, and James W. Roberts. Evaluating the number of active flows in a scheduler realizing fair statistical bandwidth sharing. In Derek L. Eager, Carey L. Williamson, Sem C. Borst, and John C. S. Lui, editors, *Proceedings of the International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS 2005, June 6-10, 2005, Banff, Alberta, Canada*, pages 217–228. ACM, 2005.

[40] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. Swift: Delay is simple and effective for congestion control in the datacenter. In Henning Schulzrinne and Vishal Misra, editors, *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020*, pages 514–528. ACM, 2020.

[41] NT Kung and Robert Morris. Credit-based flow control for ATM networks. *IEEE network*, 9(2):40–48, 1995.

[42] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John L. Hennessy, Mark Horowitz, and Monica S. Lam. The stanford dash multiprocessor. *Computer*, 25(3):63–79, 1992.

[43] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. HPCC: high precision congestion control. In Jianping Wu and Wendy Hall, editors, *Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM 2019, Beijing, China, August 19-23, 2019*, pages 44–58. ACM, 2019.

[44] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas E. Anderson. F10: A fault-tolerant engineered network. In Nick Feamster and Jeffrey C. Mogul, editors, *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013*, pages 399–412. USENIX Association, 2013.

[45] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena E. Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: a microkernel approach to host networking. In Tim Brecht and Carey Williamson, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 399–413. ACM, 2019.

[46] Paul E. McKenney. Stochastic fairness queueing. In *Proceedings IEEE INFOCOM '90, The Conference on Computer Communications, Ninth Annual Joint Conference of the IEEE Computer and Communications Societies, The Multiple Facets of Integration, San Francisco, CA, USA, June 3-7, 1990*, pages 733–740. IEEE Computer Society, 1990.

[47] Microsoft. Microsoft Azure. https://azure.microsoft.com/.

[48] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily R. Blem, Hassan M. G. Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. TIMELY: rtt-based congestion control for the datacenter. In Steve Uhlig, Olaf Maennel, Brad Karp, and Jitendra Padhye, editors, *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015, London, United Kingdom, August 17-21, 2015*, pages 537–550. ACM, 2015.

[49] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John K. Ousterhout. Homa: a receiver-driven low-latency transport protocol using network priorities. In Sergey Gorinsky and János Tapolcai, editors, *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018*, pages 221–235. ACM, 2018.

[50] The Next Platform. Flattening networks - and budgets - with 400G ethernet. https://www.nextplatform.com/2018/01/20/flattening-networks-budgets-400g-ethernet/. January 20, 2018.

[51] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, and Felipe Huici. Flowblaze: Stateful packet processing in hardware. In Jay R. Lorch and Minlan Yu, editors, *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, pages 531–548. USENIX Association, 2019.

[52] Ahmed Saeed, Varun Gupta, Prateesh Goyal, Milad Sharif, Rong Pan, Mostafa H. Ammar, Ellen W. Zegura, Keon Jang, Mohammad Alizadeh, Abdul Kabbani, and Amin Vahdat. Annulus: A dual congestion control loop for datacenter and WAN traffic aggregates. In Henning Schulzrinne and Vishal Misra, editors, *SIGCOMM '20: Proceedings of the 2020 Annual conference of the*

ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020, pages 735–749. ACM, 2020.

[53] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. Approximating fair queueing on reconfigurable switches. In Sujata Banerjee and Srinivasan Seshan, editors, *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018*, pages 1–16. USENIX Association, 2018.

[54] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In Steve Uhlig, Olaf Maennel, Brad Karp, and Jitendra Padhye, editors, *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015, London, United Kingdom, August 17-21, 2015*, pages 183–197. ACM, 2015.

[55] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In Marinho P. Barcellos, Jon Crowcroft, Amin Vahdat, and Sachin Katti, editors, *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, pages 15–28. ACM, 2016.

[56] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. Programmable packet scheduling at line rate. In Marinho P. Barcellos, Jon Crowcroft, Amin Vahdat, and Sachin Katti, editors, *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, pages 44–57. ACM, 2016.

[57] Brent Stephens and Alan L. Cox. Deadlock-free local fast failover for arbitrary data center networks. In *35th Annual IEEE International Conference on Computer Communications, INFOCOM 2016, San Francisco, CA, USA, April 10-14, 2016*, pages 1–9. IEEE, 2016.

[58] Brent Stephens, Alan L. Cox, Ankit Singla, John B. Carter, Colin Dixon, and Wes Felter. Practical DCB for improved data center networks. In *2014 IEEE Conference on Computer Communications, INFOCOM 2014, Toronto, Canada, April 27 - May 2, 2014*, pages 1824–1832. IEEE, 2014.

[59] Vojislav Đukić, Sangeetha Abdu Jyothi, Bojan Karlaš, Muhsen Owaida, Ce Zhang, and Ankit Singla. Is advance knowledge of flow sizes a plausible assumption? In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 565–580, 2019.

[60] Erico Vanini, Rong Pan, Mohammad Alizadeh, Parvin Taheri, and Tom Edsall. Let it flow: Resilient asymmetric load balancing with flowlet switching. In Aditya Akella and Jon Howell, editors, *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, pages 407–420. USENIX Association, 2017.

[61] Jim Warner. Switch buffer size. https://people.ucsc.edu/~warner/buffer.html. 2020.

[62] Robert Williams and Bahadir Erimli. Method and apparatus for performing priority-based flow control, October 18 2005. US Patent 6,957,269.

[63] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy H. Katz. Detail: reducing the flow completion time tail in datacenter networks. In Lars Eggert, Jörg Ott, Venkata N. Padmanabhan, and George Varghese, editors, *ACM SIGCOMM 2012 Conference, SIGCOMM '12, Helsinki, Finland - August 13 - 17, 2012*, pages 139–150. ACM, 2012.

[64] Shizhen Zhao, Rui Wang, Junlan Zhou, Joon Ong, Jeffrey C. Mogul, and Amin Vahdat. Minimal rewiring: Efficient live expansion for clos data center networks. In Jay R. Lorch and Minlan Yu, editors, *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, pages 221–234. USENIX Association, 2019.

[65] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale RDMA deployments. In Steve Uhlig, Olaf Maennel, Brad Karp, and Jitendra Padhye, editors, *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015, London, United Kingdom, August 17-21, 2015*, pages 523–536. ACM, 2015.

[66] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Klaus-Tycho Förster, Arvind Krishnamurthy, and Thomas E. Anderson. Understanding and mitigating packet corruption in data center networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*, pages 362–375. ACM, 2017.

A ADDITIONAL EXPERIMENTS

In this appendix, we present a more complete set of simulation results for BFC. We first summarize those results, and then present them.

Understanding the Limits of BFC: A limitation of BFC is that performance can degrade when collisions occur. The worst case is when many long-running flows share a bottleneck link with bursty traffic. We synthetically create this scenario and show that by adding a very simple end-to-end control system to BFC, we can largely ameliorate the impact of long flows, while still fully utilizing the link. See App. A.1 for details.

Comparison with Homa: Homa is a receiver driven data center transport that uses network priorities to achieve an approximation of the shortest remaining flow first (SRF) scheduling to provide low latency for short flows while still using the full bandwidth of the bottleneck for long flows. Homa also uses packet spraying. In App. A.2, we configure BFC with a similar scheduling policy. We show that Homa with packet spraying outperforms BFC, but when we turn off packet spraying, BFC outperforms Homa.

Priority Scheduling: Data center operators often classify traffic into multiple classes and use scheduling priorities to ensure performance for the most time-sensitive traffic. We repeat the experiment in Fig. 11b but with traffic split equally among four priority traffic classes, and show that BFC performs well in this case. See App. A.3 for details.

Parameter Sensitivity: We perform parameter sensitivity analysis for HPCC, DCTCP and ExpressPass. See App. A.4 for details.

Spatial Locality: We repeat the experiment in Fig. 11 with spacial locality in source-destination pairs such that the average load on all links across the network is same. The trends in performance are similar. See App. A.5 for details.

Slow-start: We evaluate the impact of using TCP slow-start instead of starting flows at line rate. We repeat the experiment in Fig. 11 and compare the original DCTCP with slow start (DCTCP + SS) and our modified DCTCP where flows start at the line rate. With incast, DCTCP + SS reduces buffer occupancy by reducing the intensity of incast flows, improving tail latency. However, it also increases median FCTs by up to 2×. Flows start at a lower rate, taking longer to ramp up to the desired rate. In the absence of incast, it increases both the tail and median FCT for short flows. See App. A.6 for details.

Reducing Contention for Queues: We tried a variant of BFC where the sender labels incast flows explicitly (similar to the potential optimization in [49]). All the incast flows at an egress port are assigned to the same queue. This frees up queues for non-incast traffic and reduces collisions substantially under large incasts. (see App. A.7).

Incremental Deployment: We repeated the experiment in Fig. 11a in the scenario where (i) BFC is deployed in part of
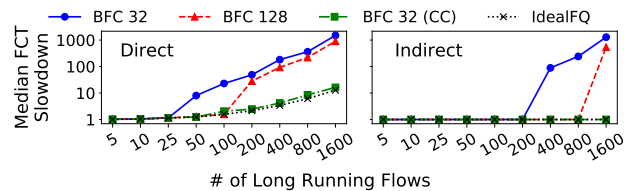
Figure 15: Median FCT slowdown for mice flows in the presence of long-running flows.

the network; (ii) The switch doesn't have enough capacity to handle all the recirculations. The impact on FCTs is minimal under these scenarios (see App. A.8).

Performance in Asymmetric Topologies: BFC makes no assumption about the topology, link speeds and link delays. We evaluate the performance of BFC in a multi-data-center topology. BFC achieves low FCT for flows within the data center, and high link utilization for the inter-data-center links (see App. A.9).

Dynamic vs. Stochastic Queue Assignment in BFC: We repeat the experiment in Fig. 11a but use stochastic hashing to statically assign flows to physical queue instead. With stochastic assignment, the number of collisions in physical queues increases, hurting FCTs (see App. A.10).

Size of Flow Table: Reducing the size of the flow table can increase index collisions in the flow table, potentially hurting FCTs. We repeat the experiment in Fig. 11a and evaluate the impact of size of flow table. Reducing the size partly impacts the short flow FCTs (see App. A.11).

Incast Flow Performance: App. A.12 shows the slowdown for incast flows for the Google workload used in Fig. 9. BFC reduces the FCT for incast flows compared to other feasible schemes.

A.1 Understanding the limits of BFC

This section investigates the impact of large numbers of active flows on BFC's performance through controlled microbenchmarks. We also show that adding a simple end-to-end flow control mechanism on top of pure BFC helps alleviate the performance impairments caused by large numbers of flows.

Collisions hurt performance in two ways. Consider a congested port X. First, at X, the packets of a short flow can get stuck behind the packets of a long flow sharing the same queue, increasing the FCT. Such performance degradation occurs when the number of active flows exceeds the number of queues at X. Second, X can pause an upstream queue. Unrelated flows sharing this upstream queue will get paused even though they are not going through the congested port X (congestion spreading). BFC can leverage the larger number of upstream queues at the upstream switches to limit congestion spreading (§3.3.1). Typically, congestion spreads only once the number of flows at the congested port exceeds the total number of upstream queues. As a result, in larger topologies with more upstream switches, congestion spreading is harder to create.

To illustrate these issues, we conduct experiments on our standard topology (§6.2.1) where we create different numbers of long-running elephant flows destined to the same receiver (Receiver *A*). All elephant flows start at the beginning of the experiment. We then create two groups of short flows: (1) destined to the same receiver *A* (referred as "direct" mice flows), and (2) destined to a different receiver *B* in the same rack as receiver *A* (referred to as "indirect" mice flows). The aggregate load for each group of mice flows is 3% of the link capacity, and the size of the mice flows is 1 KB. Fig. 15 shows the median FCT slowdown for mice flows as we vary the number of long-running flows. We show results for BFC with 32 and 128 queues, and also IdealFQ (described in §6.2.1) for reference. As expected, for direct mice flows, the FCT degrades when the number of long-running flows exceeds the number of queues. For indirect flows, the degradation only happens when long flows exceed $8\times$ the number of queues, since the topology has 8 spine switches connected to each ToR switch. In this case, some indirect mice flows get paused unnecessarily because they share an upstream queue with a paused long-running flow.

Combining end-to-end congestion control with BFC: In the previous experiment, each long-running flow can build up to 1 Hop-BDP of buffering before getting paused. With *N* long-running flows, in the worst case, a mice flow experiencing a collision can get stuck behind $N\times$ 1-Hop BDP of buffering. BFC can use a simple end-to-end congestion control mechanism to reduce this buffering and limit HoL blocking. This mechanism is helpful in scenarios with persistently large numbers of active flows. As our evaluations showed (§6.3), even in workloads with high load and occasional large-scale incast, pure BFC (with no end-to-end control) performs well except in extreme cases.

Augmenting BFC with end-to-end control is simple. The main goal of the end-to-end control is to prevent flows from sending an excessively large number of packets into the network. Importantly, the end-to-end mechanism need not try to accurately control queuing, react quickly to bursts, or achieve fairness — typical requirements for low-latency data center congestion control protocols — since BFC already achieves these goals.

As an example, we implemented a simple delay-based congestion control that tries to maintain the end-to-end RTT at a certain threshold (RTT_{Target}). We chose a high RTT_{Target} value of $2.5\times$ base RTT to avoid hurting the throughput of long flows, exploiting the fact that it isn't necessary to tightly control queuing in BFC. The algorithm adjusts the sender's window (*w*) as follows.

With the above rule, the window of a sender roughly goes from $w \rightarrow w \times \frac{\text{RTT}_{Target}}{RTT}$ within an RTT. Fig. 15 shows the performance with this variant (BFC 32 (CC)). The performance is close to IdealFQ in all the cases. To check if this change negatively affected the overall behavior of BFC, we repeat the principle experiment in Fig. 11 (Facebook work-

$$\text{RTT}_{Target} = 2.5\times \text{ Base RTT};$$
$$w = 1 \text{ BDP};$$
for *each Acknowledgement* **do**
  if $RTT > RTT_{Target}$ **then**
    $w = w - \frac{\text{RTT} - \text{RTT}_{Target}}{RTT}$
  else
    $w = w + \frac{\text{RTT}_{Target} - RTT}{RTT}$

Algorithm 1: Simple end-to-end congestion control

(a) 55% + 5% 100-1 incast   (b) 60%
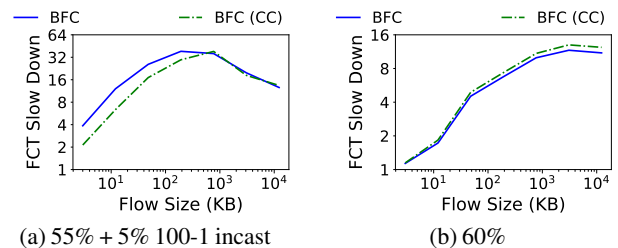
Figure 16: 99th percentile FCT slowdown when combined with congestion control. Facebook workload, same setup as Fig. 11.

load) with BFC 32 (CC). Fig. 16 shows the 99th percentile FCT slowdowns. The FCTs of long flows are similar to that of the original BFC (within 10%). However, in the presence of incast, adding congestion control improves the 99th percentile FCT of short flows and the peak buffer occupancy by 30%. While using end-to-end congestion control can improve performance under frequent collisions (and we advocate supplementing BFC with such a mechanism in practice), in this paper we focus on BFC without any such mechanism to better understand the core benefits and limitations of BFC in its purest form.

In App. A.7, we experiment with a variant of BFC where the sender labels incast flows explicitly (similar to the potential optimization in [49]). All the incast flows at an egress port are assigned to the same queue. This frees up queues for non-incast traffic and reduces collisions substantially under large incasts.

A.2 Comparison with Homa

Homa is a receiver driven data center transport that uses network priorities to achieve an approximation of shortest-remaining-flow-first (SRF) scheduling. Homa divides a flow's data into unscheduled (first BDP of traffic) and scheduled categories. The sender assigns a fixed priority level to a flow's unscheduled bytes based on its size and the flow size distribution of the workload. The unscheduled bytes are transmitted at line rate. The receiver assigns priority levels to the scheduled bytes and issues grants (credits) for them. Homa assumes per-packet spraying to ensure load balancing across core links, and sufficient core capacity to guarantee minimal congestion in the core.

While we focus on fair queuing in this paper, BFC's design is applicable to other scheduling policies. In this section, we
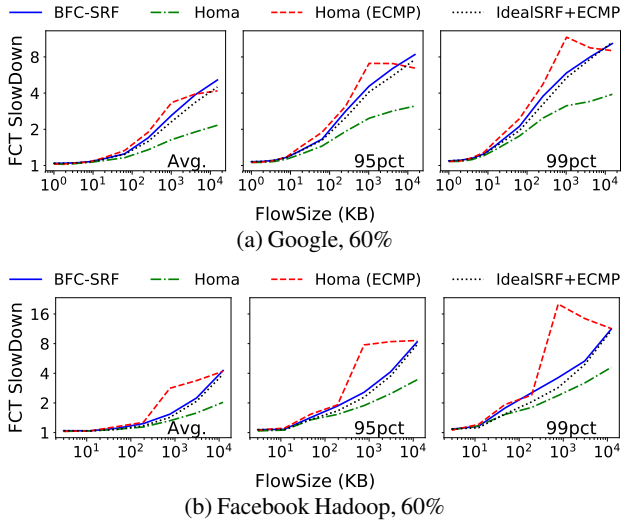
(a) Google, 60%

(b) Facebook Hadoop, 60%

Figure 17: FCT slowdown on an oversubscribed clos topology. With packet spraying, Homa encounters minimal congestion in the core and outperforms other schemes.

Scheme	Link	95% Delay (μs)	99% Delay (μs)
Homa	Agg-ToR	2.4	6.7
Homa	ToR-Agg	2.1	6.0
Homa ECMP	Agg-ToR	40.8	87.2
Homa ECMP	ToR-Agg	43.7	93.3

Table 2: Per-packet queuing delay for scheduled traffic in the core.

(a) Google, 60%

(b) Facebook Hadoop, 60%

Figure 18: BFC's dynamic queueue assignment achieves a better approximation of the SRF scheduling policy. BFC-SRF achieves close to optimal FCTs.

evaluate a variant of BFC, BFC-SRF, that aims to approximate SRF. Flows insert their remaining size into a header field in each packet transmitted, and the switch schedules queues in order of remaining size of the packet at the head of the queue. Similarly to Homa, NICs also follow SRF scheduling. We ran Homa using its OMNet++simulator [2]. The Homa simulator assumes unbounded buffers at the switch. For BFC, we use a 12 MB shared buffer. We use 32 queues for both Homa and BFC. For Homa, the 32 priority levels are divided between unscheduled and scheduled priorities based on the ratio of unscheduled and scheduled traffic; the overcommitment level is equal to the number of scheduled priorities [49]. We use our default topology with 128 servers and 2:1 oversubscription at the ToR uplinks (§6.2.1).

Two differences between Homa and BFC-SRF are worth highlighting. First, BFC-SRF uses flow-level ECMP rather than packet spraying for enforcing per-flow backpressure. Second, BFC-SRF uses dynamic queue assignment and performs SRF scheduling directly on the switch, as opposed to Homa's priority assignment from the end-points. To understand the impact of these aspects separately, we also evaluate a variant of Homa with ECMP, and report results for IdealSRF+ECMP, an idealized SRF scheme with unlimited queues and unbounded buffers at each switch with ECMP load balancing.

We repeat the experiments in Fig. 10 and Fig. 11b for the Google and Facebook workloads at 60% load (log-normal flow arrivals without incast). Fig. 17 reports the FCTs. Homa performs the best out of all schemes, achieving up to 2$\times$ better FCTs for long flows. With packet spraying, flows encounter minimal congestion in the core, and compete for bandwidth primarily at the last-hop. In contrast, ECMP is prone to path collisions [7] and flows encounter congestion in the core. Notice that a last-hop link carries half the load of a core
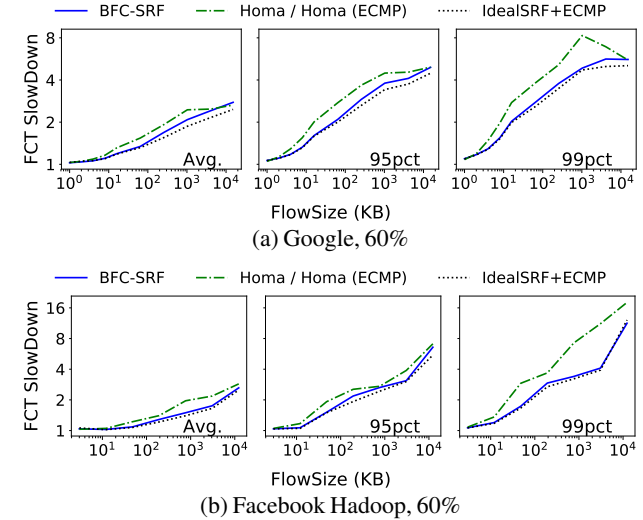
link (30% vs 60%) in this experiment on average (§6.2.1). Since packet spraying essentially eliminates congestion on the core links, with Homa flows experience congestion only on the last-hop links. But with the ECMP-based schemes, flows contend at the core links (with 2$\times$ the load). As a result, Homa even outperforms IdealSRF+ECMP. This result illustrates the benefits of packet spraying; nevertheless, packet spraying is rarely deployed in practice because it can cause packet reordering, increasing CPU overhead at endpoints[7], and it can hurt performance in asymmetric topologies (e.g., caused by rolling upgrades or link failures) [60].

Among the ECMP approaches, BFC-SRF is close to IdealSRF+ECMP and Homa is worse. In Homa, receivers have no visibility into congestion in the core and don't react to queue buildup in the core (though each flow limits its total in-flight data to 1 BDP). Also, Homa's receiver-set priorities are only based on contending flows at the last hop, and can violate SRF scheduling when congestion occurs in the core. Table 2 shows that with ECMP, the scheduled traffic encounters significantly higher queuing in the core.

Benefits of BFC's dynamic queue assignment over Homa. BFC makes queue assignment and scheduling decisions at the switch, based on an instantaneous view of competing flows. In principle, this should allow BFC to more accurately approximate SRF compared to Homa. To understand if this

---

[7]Packet reordering makes hardware offloads such as Large Receiver Offload (LRO) ineffective [26].

(a) Google, 55% + 5% 100-1 incast

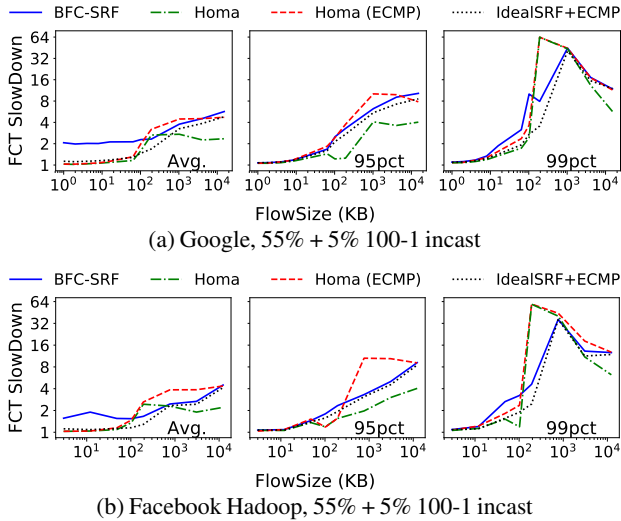(b) Facebook Hadoop, 55% + 5% 100-1 incast

Figure 19: FCT slowdown with 100-1 incast. Collisions in BFC-SRF can cause priority inversions hurting FCTs

is actually the case, we conduct an experiment with the same Google and Facebook workloads but with all flows destined to a single receiver, and the senders located within the same rack as the receiver. Since there is no traffic in the core, load balancing (ECMP vs. packet spraying) does not matter in this case. Flow arrivals are log-normal and the load on the receiver's link is 60%. Fig. 18 shows the results. BFC-SRF achieves better FCTs primarily at the tail.

We give two examples of priority inversions in Homa which BFC avoids. First, the Homa sender assigns priorities to unscheduled traffic based on flow size distributions rather than using the current set of flows competing at the switch due to lack of visibility for the first RTT. As a result with Homa, short flows (< 1 BDP) with similar flow sizes can end up sharing unscheduled priority queues unnecessarily, even when there are sufficient queues at the switch to assign each flow a unique queue. Second, in Homa the unscheduled bytes of a flow are always scheduled ahead of the scheduled bytes of competing flows. This implies that the unscheduled bytes of a new long flow will be *incorrectly* scheduled ahead of the scheduled bytes of a shorter flow. This also violates SRF and increases FCT for flows larger than a BDP.

Impact of collisions on BFC-SRF. Recall that with large incast, BFC can experience collisions. For BFC-SRF, such collisions can cause priority inversions that hurt FCTs. To illustrate this, we repeat the experiments in Fig. 9 and Fig. 11a (55% load plus 5% 100-1 incast traffic). Fig. 19 shows that the average FCT for short flows is higher with BFC-SRF. This is because of high completion times for a (small) fraction of short flows sharing queues with longer flows. To understand why, consider the following situation. An incoming short flow arrives when there are no free queues, and ends up sharing the queue with a long flow. Let's say the remaining size of the long flow is greater than the incast flow size (200 KB in

this experiment). In case there are competing incast flows present in other queues, the incast flows will be scheduled ahead of this long flow. Therefore, the short flow will have to wait for *all* the traffic from the incast flows to finish to make any progress. This can severely degrade its completion time. The core of this problem is that when a port runs out of queues, the BFC switch assigns the new flow to a queue randomly. This is fine for fair queuing but with SRF, a more sophisticated strategy may improve performance (e.g., assign the new flow to a queue with similar remaining flow sizes).

As explained earlier, Homa is not immune to priority inversions. Fig. 19 shows that with Homa, flows with size greater than 1 BDP but less than 2 BDP have high FCTs at the tail. This is because unscheduled bytes of the the incast flows are incorrectly scheduled ahead of the scheduled bytes of such flows.

These experiments suggest an interesting possibility to try to get the best of both schemes: we could combine BFC's dynamic queue assignment for unscheduled traffic with Homa's grant mechanism for controlling scheduled traffic. We leave exploration of such a design to future work.

A.3 Multiple traffic classes

Many data center operators allocate network traffic into a small number of priority traffic classes to ensure that mission critical traffic is delivered with low tail latency, while other traffic is delivered according to its quality of service needs. BFC has a simple extension to support priority groups. To avoid priority inversion where a flow at one priority can be stalled behind a flow of a lower priority, we assume queues at a port are statically assigned to different priority levels. The switch performs dynamic queue assignment for each class independently. A flow with priority X is only assigned to physical queues associated with that priority. Queues at the same priority level follow fair scheduling.

Statically partitioning physical queues among traffic classes could make it more likely for traffic within a class to run out of queues and suffer degraded performance with collisions and HoL blocking. On the other hand, high priority traffic is preferentially scheduled, leading to short queues and few active flows. Collisions will be more likely at lower priority traffic classes, where performance is already degraded. Priority scheduling results in rapid and extreme changes in the available rate for these background classes. Relative to end-to-end control, per-hop backpressure can more easily utilize rapidly changing spare capacity.

To test how BFC behaves with multiple traffic classes, we repeat the experiment in Fig. 11b: Facebook workload, 60% load, and no incast. We configure the system with 4 priority classes, each with equal load (15% each, 60% in aggregate). We allocate physical queues evenly to each traffic class. We consider configurations with 32 and 128 queues per port (8 or 32 queues per class). We also show results for HPCC and DCTCP. In this study, DCTCP marks packets based on per-
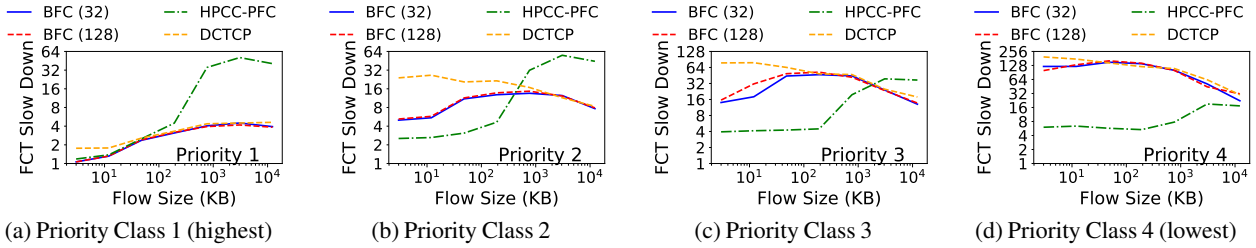
(a) Priority Class 1 (highest)   (b) Priority Class 2   (c) Priority Class 3   (d) Priority Class 4 (lowest)

Figure 20: Multiple traffic classes with BFC, reporting 99th percentile FCT slowdown for the Facebook workload, 60% load, and no incast.

class queueing, while HPCC uses switch aggregates. Fig. 20 shows the 99th percentile FCT slowdown for different priority classes. BFC achieves good performance across all traffic classes and flow sizes. In particular, BFC achieves up to $5\times$ better tail latency for short flows than DCTCP. At the lowest priority level, DCTCP's short flow tail latency converges to that of BFC. For low priority flows, tail latency is primarily governed by time spent waiting to be scheduled at the switch.

HPCC's performance is somewhat anomalous. Long flows suffer priority inversion, where long flows at high priority achieve significantly worse service than short flows at lower priority. In HPCC, long flows back off in an attempt to keep queues empty. The (transient) extra capacity left by such long flows can be used by short flows traffic at all priority levels, improving performance for these short flows.

BFC has only slightly better performance with 32 vs. 8 queues per priority level, indicating that collisions did not have much impact. For high priority traffic, the setup is equivalent to running our experiment with just one traffic class at 15% load and a small number of queues—even modest numbers of active queues are unlikely at such low load. Lower priority traffic can run out of queues, but they gain the benefit of being able to take immediate advantage when the high priority queues are empty. In other words, work conserving behavior is more important for background traffic than the number of queues. We acknowledge this is just one study, and there are likely scenarios where BFC's performance could suffer when using multiple traffic classes.

One obvious improvement is to split queues dynamically among classes rather than statically. But in the long run, we strongly believe that the number of queues per port is likely to continue to grow to whatever is needed to deliver good performance.

A.4 Parameter sensitivity for comparison schemes

In this section, we perform sensitivity analysis to understand the impact of parameters on performance of HPCC, DCTCP and ExpressPass. We repeat the experiment in Fig. 11b (Facebook distribution with 60% load). Fig. 21 reports the average, 95th and 99th percentile flow completion times as we vary the parameters. In general, we observe that parameters present a trade-off between the latency of short flows (queuing) and the throughput of long flows (link utilization).

HPCC: We vary the target utilization (η) from 90 to 98%.

(a) HPCC (η)

(b) DCTCP (ECN marking threshold: K_{min}-K_{max})

(c) ExpressPass (Credit Buffer Size)
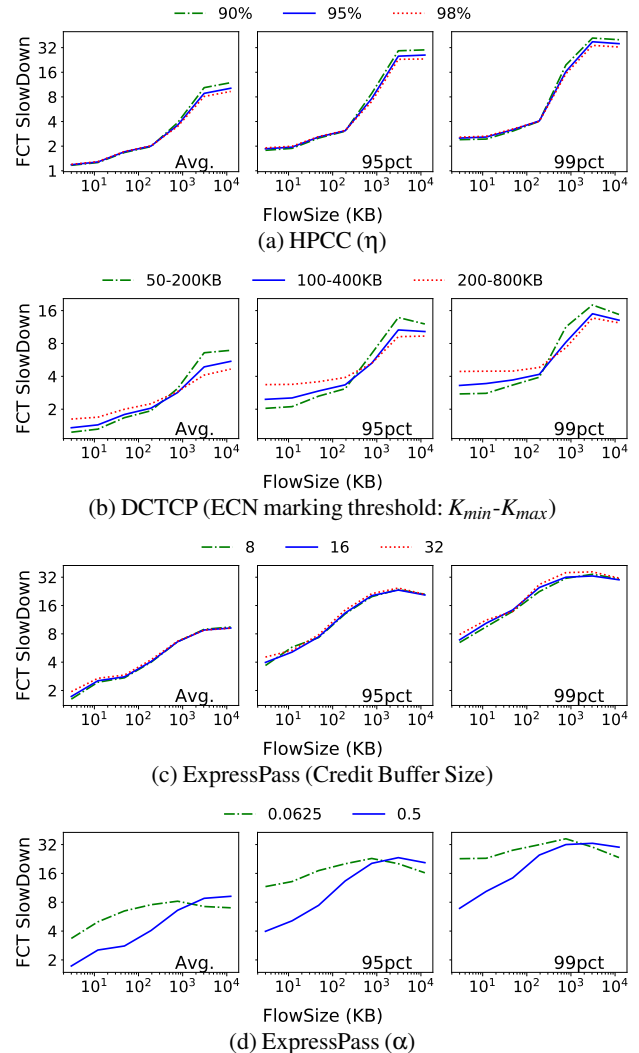
(d) ExpressPass (α)

Figure 21: 99th percentile FCT slowdown for the Facebook workload, 60% load without incast. Sensitivity to the choice of parameters in HPCC, DCTCP, and ExpressPass.

As expected, increasing η worsens the FCT of short flows but improves the FCT for long flows (marginally for both), see Fig. 21a.

DCTCP: We vary the ECN marking threshold governed by parameters K_{min} and K_{max}. Increasing the threshold increases the queuing at the switch, which increases FCT of short flows but improves link utilization (Fig. 21b).
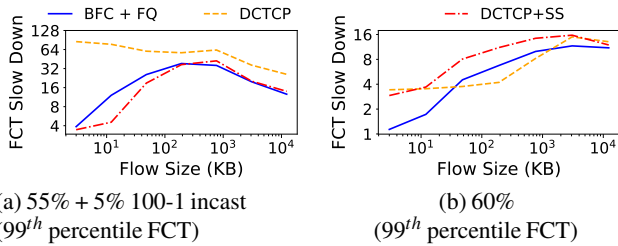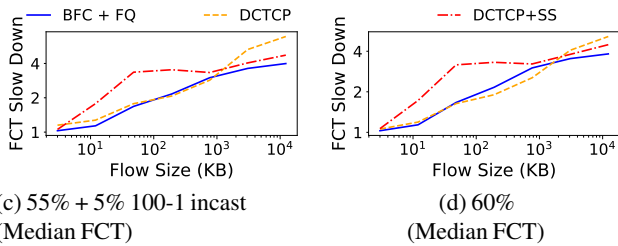
(a) 55% + 5% 100-1 incast   (b) 60%
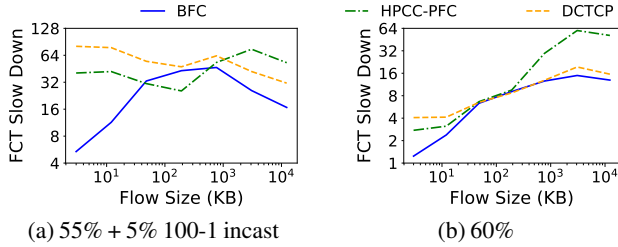
Figure 22: Impact of spatial locality. FCT slowdown (99th percentile) for Facebook distribution with and without incast.

(a) 55% + 5% 100-1 incast
(99th percentile FCT)

(b) 60%
(99th percentile FCT)

(c) 55% + 5% 100-1 incast
(Median FCT)

(d) 60%
(Median FCT)

Figure 23: Impact of using slow start on median and 99th percentile tail latency FCT slowdown, for the Facebook flow size distribution with and without incast (setup the same as Fig. 11). With incast, DCTCP + SS (slow start) reduces the tail FCT, but it increases median FCTs by up to 2 ×. In the absence of incast, DCTCP + SS increases both the tail and median FCT for short and medium flows.

ExpressPass: Varying the credit buffer size has little impact on performance (Fig. 21c). We vary α, which controls how the receiver credits are generated. Reducing α reduces "credit waste", improving the FCT of long flows. However, it also increases the FCT of short flows (Fig. 21d).

A.5   Impact of Spatial Locality

We repeated the experiment from Fig. 11 with spatial locality in source-destination pairs such that the average load on all links across the network is same. Fig. 22 shows the 99th percentile slowdowns. The trends are similar to Fig. 11.

A.6   Using TCP Slow-start

We also evaluate the impact of using TCP slow-start instead of starting flows at line rate in Figure 23. We compare the original DCTCP with slow start (DCTCP + SS) with an initial window of 10 packets versus the modified DCTCP used so far (initial window of the BDP). The setup is same as Fig. 11.

With incast, DCTCP + SS reduces buffer occupancy by reducing the intensity of incast flows, improving tail latency (Fig. 23a). However, slow start increases the median FCT substantially (Fig. 23c). Flows start at a lower rate, taking

(a) Average FCT for long flows   (b) Tail FCT for short flows
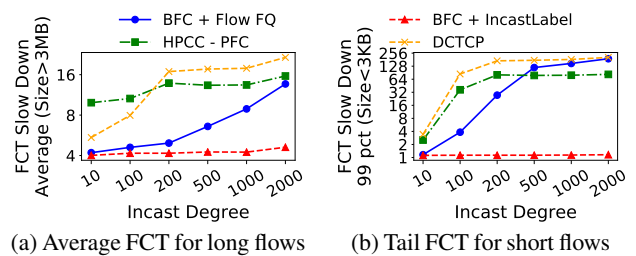
Figure 24: FCT slowdown for short and long flows as a function of incast degree. The x axis is not to scale. By isolating incast flows, BFC + IncastLabel reduces collisions and achieves the best performance.

longer to ramp up to the desired rate. For applications with serially dependent flows, an increase in median FCTs can impact the performance substantially.

In the absence of incast, slow start increases both the tail (Fig. 23b) and median (Fig. 23d) FCT for the majority of flow sizes. In particular, short flows are still slower than with BFC, as slow start does not remove burstiness in buffer occupancy in the tail.

A.7   Reducing contention for queues

To reduce contention for queues under incast, we tried a variant of BFC where the sender labels incast flows explicitly (similar to the potential optimization in [49]). BFC + IncastLabel assigns all the incast flows at an egress port to the same queue. This frees up queues for non-incast traffic, reducing collisions and allowing the scheduler to share the link between incast and non-incast traffic more fairly.

Fig. 24 shows the performance of BFC + IncastLabel in the same setup as Fig. 13. The original BFC is shown as BFC + Flow FQ for per-flow fair queuing. BFC + IncastLabel achieves the best performance across all the scenarios. However, the FCTs for incast flows is higher compared to BFC + Flow FQ (numbers not shown here). When there are multiple incast flows at an ingress port, the incast flows are allocated less bandwidth in aggregate compared to per-flow fair queuing.

While BFC + IncastLabel achieves great performance, it assumes the application is able to label incast flows, and so we use a more conservative design for the main body of our evaluation.

A.8   Incremental Deployment

We repeated the experiment in Fig. 11a in the scenario where i) BFC is deployed in part of the network; ii) The switch doesn't have enough capacity to handle all the recirculations. Fig. 25 reports the tail FCT and buffer occupancy for these settings.

Partial deployment in the network: We first evaluate the situation when BFC is only deployed at the switches and the sender NICs don't respond to backpressure signal (shown as BFC - NIC). To prevent sender NIC traffic from filling up the buffers at the ToR, we assume a simple end-to-end congestion control strategy where the sender NIC caps the in-flight packets for a flow to 1 end-to-end bandwidth
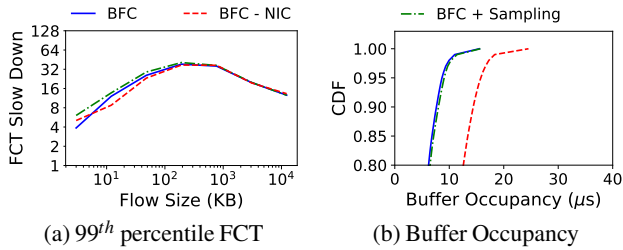
(a) 99th percentile FCT   (b) Buffer Occupancy

Figure 25: FCT slowdown (99th percentile) and buffer occupancy distribution for two BFC variants. When NICs don't respond to backpressure (BFC - NIC), BFC experiences moderate increased buffering. Using sampling to reduce recirculation (BFC + sampling) has marginal impact on performance.

delay product (BDP). As expected, BFC - NIC experiences increased buffering at the ToR (Fig. 25b). However, the tail buffer occupancy is still below the buffer size and there are no drops. Since all the switches are BFC enabled and following dynamic queue assignment, the frequency of collisions and hence the FCTs are similar to the orignal BFC.

Sampling packets to reduce recirculations: A BFC switch with an RMT architecture [18] recirculates packets to execute the dequeue operations at the ingress port. Depending on the packet size distribution of the workload, a switch might not have enough packet processing (pps) capacity or recirculation bandwidth to process these recirculated packets. In such scenarios, we can reduce recirculations by sampling packets. Sampling works as follows.

On a packet arrival (enqueue), sample to decide whether a packet should be recirculated or not. Only increment the pause counter and `size` in the flow table for packets that should be recirculated. The dequeue operations remain as is and are only executed on the recirculated packets. The `size` now counts the packets sampled for recirculation and residing in the switch. While sampling reduces recirculations, it can cause packet re-ordering. Recall, BFC uses `size` to decide when to reassign a queue. With sampling, `size` can be zero even when a flow has packets in the switch. This means a flow's queue assignment can change when it already has packets in the switch, causing reordering. However, sticky queue assignment should reduce the frequency of these events (§3.3.2).

We now evaluate the impact of sampling on the performance of BFC (shown as BFC + Sampling). In the experiment, the sampling frequency is set to 50%, i.e., only 50% of the packets are recirculated. BFC + Sampling achieves nearly identical tail latency FCT slowdowns and switch buffer occupancy as the orignal BFC. With sampling, fewer than 0.04% of the packets were retransmitted due to packet reordering.

A.9 Cross data center traffic

For fault tolerance, many data center applications replicate their data to nearby data centers (e.g., to a nearby metro area). We evaluate the impact of BFC on managing cross-data center congestion in such scenarios. We consider the ability

(a) 99th percentile FCT   (b) Utilization at the interconnect.
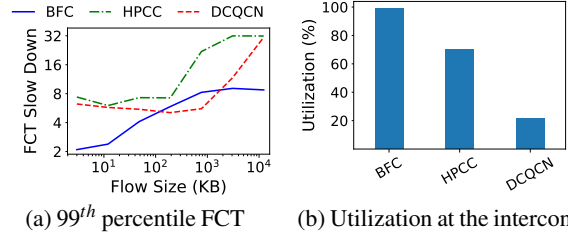
Figure 26: Performance in cross data center environment where two data center are connected by a 200 μs link, for the Facebook workload (60% load) with no incast traffic. The left figure shows the 99th percentile FCT slowdown for intra-data-center flows. The right figure shows the average utilization of the link connecting the two data centers.

of different systems to achieve good throughput for the inter-data-center traffic, and we also consider the impact of the cross-data-center traffic on tail latency of local traffic, as the larger bandwidth-delay product means more data is in-flight when it arrives at the bottleneck.

We created a Clos topology with 64 leaf servers, and 100 Gbps links and 12 MB switch buffers. Two gateway switches connect the data centers using a 200 Gbps link with 200 μs of one-way delay (i.e. the base round trip delay of the link is 400 μs), or roughly equivalent to the two data centers being separated by 50 km assuming a direct connection. The experiment consists of intra-data-center flows derived from the Facebook distribution (60% load). Additionally, there are 20 long-lived inter-data-center flows in both the directions.

Fig. 26a shows the 99th percentile tail latency in FCT slowdown for intra-data-center flows for BFC, HPCC and DCQCN.[8] Fig. 26b shows the average utilization of the link connecting the two data centers (interconnect), a proxy for the aggregate throughput of the long-lived inter-data-center flows. BFC is better for both types of flows. With BFC, the link utilization of the wide area interconnect is close to 100%, while neither HPCC nor DCQCN can maintain the link at full utilization, even with ample parallelism. This is likely a consequence of slow end-to-end reaction of the inter-data-center flows [52]. The congestion state on the links within a data center is changing rapidly because of the shorter intra-data-center flows. By the time an inter-data-center flow receives congestion feedback and adjusts its rate, the congestion state in the network might have already changed. When capacity becomes available, the inter-data-center flows can fail to ramp up quickly enough, hurting its throughput.

Relative to the single data center case (cf. Fig. 11b), tail latency FCTs are worse for all three protocols, but the relative advantage of BFC is maintained. Where HPCC has better tail latency than DCQCN in the single data center case for both short and medium-sized flows, once inter-data-center traffic is added, HPCC becomes worse than DCQCN. With bursty workloads, on the onset of congestion, the long-lived

---

[8]Data center operators have developed specialized protocols for better inter-data center link management [21]; comparing those to BFC is future work.

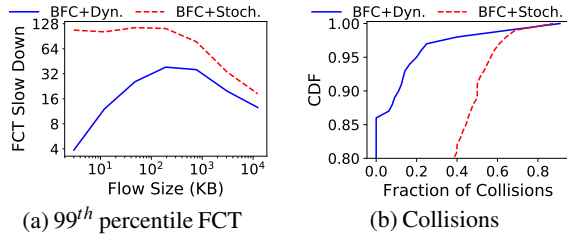(a) 99th percentile FCT   (b) Collisions

Figure 27: Performance of BFC with stochastic queue assignment, for the workload in Fig. 11a. BFC + Stochastic incurs more queue collisions leading to worse tail latency especially for small flows compared to BFC + Dynamic.

flow will take an end-to-end RTT to reduce its rate, and can build up to 1 BDP (or 500 KB) of buffering, hurting the tail latency of intra-data-center traffic. This has less of an impact on DCQCN because it utilizes less of the inter-data-center bandwidth in the first place.

In contrast, BFC reacts at the scale of the hop-by-hop RTT. Even though inter-data-center flows have higher end-to-end RTTs, on switches within the data center, BFC will pause/resume flows on a hop-by-hop RTT timescale ($2\,\mu$s). As a result, with BFC, tail latencies of intra-data-center flows are relatively unaffected by the presence of inter-data-center flows, while the opposite is true of HPCC.

A.10   Physical queue assignment

To understand the importance of dynamically assigning flows to physical queues, we repeated the experiment in Fig. 11a with a variant of BFC, BFC + Stochastic, where we use stochastic hashing to statically assign flows to physical queues (as in SFQ). In BFC (referred as BFC + Dynamic here), the physical queue assignment is dynamic. To isolate the effect of changing the physical queue assignment, the pause thresholds are the same as BFC + Dynamic.

Fig. 27a shows the tail latency. Compared to BFC, tail latency for BFC + Stochastic is much worse for all flow sizes. Without the dynamic queue assignment, flows are often hashed to the same physical queue, triggering HoL blocking and hurting tail latency, even when there are unoccupied physical queues. Fig. 27b is the CDF of such collisions. BFC+Stochastic experiences collisions in a high fraction of cases and flows end up being paused unnecessarily. Such flows finish later, further increasing the number of active flows and collisions. Even with incast, the number of active flows in BFC is smaller than the number of physical queues most of the time.

A.11   Size of flow table

We repeated the experiment in Fig. 11a, but varied the size of the flow table (as a function of the number of queues in the switch). The default in the rest of the paper uses a flow table of 100X. Fig. 28 shows the tail latency as a function of flow size, for both smaller and larger flow tables. Reducing the size of the flow table increases the index collisions in the flow

(a) 99th percentile FCT
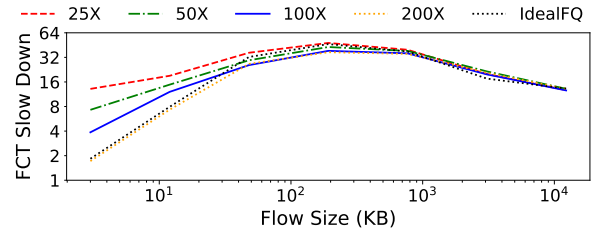
Figure 28: FCT slowdown (99th percentile) for BFC for different size flows as a function of the size of the flow table (as a multiple of the number of queues in the switch). The other experiments in the paper use a flow table of 100X. Further reducing the size of the flow table hurts small flow performance.
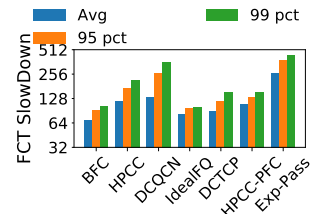
Figure 29: FCT slowdown for incast traffic. Slowdown is defined per flow. BFC reduces the FCT for incast flows compared to other feasible schemes. Setup from Fig. 9.

table. Each flow table collision means that those flows are necessarily assigned to the same physical queue. Tail latency FCTs degrade as a result, particularly for small flows and for smaller table sizes. This experiment shows that increasing the size of the flow table would moderately improve short flow tail latency for BFC.

A.12   Incast flow performance

Fig. 29 shows the slowdown for incast flows for the Google workload used in Fig. 9. The benefits of BFC for non-incast traffic do not come at the expense of worse incast performance. Indeed, BFC improves the performance of incast flows relative to end-to-end congestion control, because it reacts faster when capacity becomes available at the bottleneck, reducing the percentage of time the bottleneck is unused while the incast is active.

B   DEADLOCK PREVENTION

We formally prove that BFC is deadlock-free in absence of cyclic buffer dependency. Inspired by Tagger [32], we define a backpressure graph ($G(V,E)$) as follows:

1. Node in the graph (V): A node is an egress port in a switch and can thus be represented by the pair <switchID, egressPort>.
2. Edge in the graph (E): There is a directed edge from $B \rightarrow A$, if a packet can go from A to B in a single hop (i.e., without traversing any other nodes) and trigger backpressure from $B \rightarrow A$. Edges represent how backpressure can propagate in the topology.

We define deadlock as a situation when a node (egress port) contains a queue that has been paused indefinitely.

Cyclic buffer dependency is formally defined as the situation when G contains a cycle.

Theorem 1 *BFC is deadlock-free if $G(V,E)$ does not contain any cycles.*

Proof: We prove the theorem by using contradiction.

Consider a node A that is deadlocked. A must contain a queue (A_q) that has been indefinitely paused as a result of backpressure from the downstream switch. If all the packets sent by A_q were drained from the downstream switch, then A_q will get unpaused (§3.3.2). There must be at least one node (B) in the downstream switch that triggered backpressure to A_q but hasn't been able to drain packets from A_q, i.e., B is deadlocked. This implies, in G, there must be an edge from $B \rightarrow A$. Applying induction, for B there must exist another node C (at the downstream switch of B) that is also deadlocked (again there must be an edge from $C \rightarrow B$). Therefore, there will be an infinite chain of nodes which are paused indefinitely, the nodes of the chain must form a path in G. Since G doesn't have any cycles, the paths in G can only be of finite length, and therefore, the chain cannot be infinitely long. A contradiction, hence proved.

Preventing deadlocks: To prevent deadlocks, given a topology, we calculate the backpressure graph, and pre-compute the edges that should be removed so that the backpressure graph doesn't contain any cycles. Removing these edges thus guarantees that there will be no deadlocks even under link failures or routing errors. To identify the set of edges that should be removed we can leverage existing work [32].

To remove a backpressure edge $B \rightarrow A$, we use the simple strategy of skipping the backpressure operation for packets coming from A going to B at the switch corresponding to B.[9] Note that, a switch can identify such packets *locally* using the ingress and egress port of the packet. This information can be stored as a match-action-table (indexed by the ingress and egress port) to check whether we should execute the backpressure operations for the packet.

For Clos topologies, this just includes backpressure edges corresponding to packets that are coming from a higher layer and going back to a higher layer (this can happen due to rerouting in case of link failures). Note that, usually the fraction of such packets is small ($< 0.002\%$ [32]), so forgoing backpressure for a small fraction of such packets should hurt performance marginally (if at all).

C   IMPACT OF PAUSE THRESHOLD

A consequence of the simplicity of BFC's backpressure mechanism is that a flow can temporarily run out of packets

---

[9]To remove backpressure edges in PFC, Tagger uses a more complex approach that invloves creating new cycle free backpressure edges corresponding to the backpressure edges that should be removed. To ensure losslessness, Tagger generates backpressure using these new cycle free edges instead of the original backpressure edge. In our proposed solution, we forgo such requirement for simplicity.

at a bottleneck switch while the flow still has packets to send. The pause threshold (Th) governs the frequency of such events. Using a simple model, we quantify the impact of Th.

Consider a long flow f bottlenecked at a switch S. To isolate the impact of the delay in resuming, we assume that f is not sharing a queue with other flows at S or the upstream switch. Let μ_f be the dequeue rate of f at S, i.e., when f has packets in S, the packets are drained at a steady rate of μ_f. Similarly, let $\mu_f \cdot x$ be the enqueue rate of f at the switch, i.e., if f is not paused at the upstream, S receives packets from f at a steady rate of $\mu_f \cdot x$. Here, x denotes the ratio of enqueue to dequeue rate at S. Since f is bottlenecked at S, $x > 1$.

We now derive the fraction of time in steady state that f will not have packets in S. We show that this fraction depends only on x and Th, and is thereby referred as $E_f(x,Th)$.

The queue occupancy for f will be cyclic with three phases.
- Phase 1: S is receiving packets from f and the queue occupancy in increasing.
- Phase 2: S is *not* receiving packets from f and the queue is draining.
- Phase 3: S is not receiving packets from f while the queue is empty.

The time period for phase 1 (t_{p1}) can be calculated as follows. The queue occupancy at start of the phase is 0 and S is receiving packets from f. f gets paused when the queue occupancy exceeds Th. The queue builds at the rate $\mu_f \cdot x - \mu_f$ (enqueue rate - dequeue rate). The pause is triggered after $\frac{Th}{\mu_f \cdot (x-1)}$ time from the start of the phase. Since the pause takes an $HRTT$ to take effect, the queue grows for an additional $HRTT$. t_{p1} is therefore given by:

$$t_{p1} = \frac{Th}{\mu_f \cdot (x-1)} + HRTT. \quad (1)$$

The queue occupancy at the end of phase 1 is $Th + HRTT \cdot \mu_f \cdot (x-1)$. The time period for phase 2 (t_{p2}) corresponds to the time to drain the queue. t_{p2} is given by:

$$t_{p2} = \frac{Th + HRTT \cdot \mu_f \cdot (x-1)}{\mu_f}. \quad (2)$$

At the end of phase 2, there are no packets from f in S. As a result, S resumes f at the upstream. Since the resume takes an $HRTT$ to take effect, the queue is empty for an $HRTT$. Time period for phase 3 (t_{p3}) is given by:

$$t_{p3} = HRTT \quad (3)$$

Combining the equations, $E_f(x,Th)$ is given by:

$$E_f(x,Th) = \frac{t_{p3}}{t_{p1} + t_{p2} + t_{p3}}$$
$$= \frac{x-1}{\frac{Th}{HRTT \cdot \mu_f} \cdot x + (x^2-1)}. \quad (4)$$
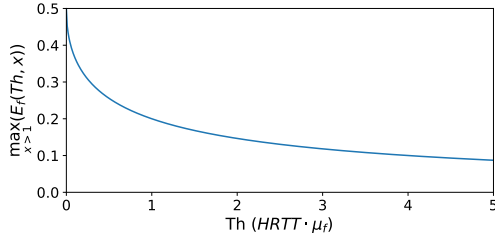
Figure 30: Impact of pause threshold (Th) on the metric of worst case inefficiency. Increasing Th reduces the maximum value for the fraction of time f can run out of packets at the bottleneck.

Notice that for a given x, $E_f(x,Th)$ reduces as we increase Th. Increasing Th, increases the time period for phase 1 and phase 2, and the fraction of time f runs out of packets reduces as a result.

We now quantify the impact of pause threshold on the worst case (maximum) value of $E_f(x, Th)$. Given a Th, $E_f(x,Th)$ varies with x. When $x \to 1$, $(E_f(x,Th) \to 0)$, and when $x \to \infty$, $(E_f(x,Th) \to 0)$. The maxima occurs somewhere in between. More concretely, for a given value of Th, the maxima occurs at $x = \sqrt{\frac{Th}{HRTT \cdot \mu_f}} + 1$. The maximum value $(\max_{x>1}(E_f(x,Th)))$ is given by:

$$\max_{x>1}(E_f(x,Th)) = \frac{1}{\left(\sqrt{\frac{Th}{HRTT \cdot \mu_f}} + 1\right)^2 + 1}. \qquad (5)$$

Fig. 30 shows how $\max_{x>1}(E_f(x,Th))$ changes as we increase the pause threshold. As expected, increasing the pause threshold reduces $\max_{x>1}(E_f(x,Th))$. However, increasing the pause threshold has diminishing returns. Additionally, increasing Th increases the buffering for f (linearly).

In BFC, we set Th to 1-Hop BDP at the queue drain rate, i.e., $Th = HRTT \cdot \mu_f$. Thereoofore, the maximum value of $E_f(x, Th)$ is 0.2 (at $x = 2$). This implies, under our assumptions, that a flow runs out of packets at most 20% of the time due to the delay in resuming a flow.

Note that 20% is the maximum value for $E_f(x,Th)$. When $x \neq 2$, $E_f(x,Th)$ is lower. For example, when $x = 1.1$ (i.e., the enqueue rate is 10% higher than the dequeue rate), $E_f(x,Th)$ is only 7.6%.

The above analysis suggests that the worst-case under-utilization caused by delay in resuming is 20%. Note that in practice, when an egress port is congested, there are typically multiple flows concurrently active at that egress. In such scenarios, the under-utilization is much less than this worst-case bound, because it is unlikely that all flows run out of packets at the same time. As our evaluation shows, with BFC, flows achieve close to ideal throughput in realistic traffic scenarios (§6).

Packet Order Matters!
Improving Application Performance by Deliberately Delaying Packets

Hamid Ghasemirahni[1], Tom Barbette[1], Georgios P. Katsikas[1],
Alireza Farshin[1], Amir Roozbeh[1,2], Massimo Girondi[1], Marco Chiesa[1],
Gerald Q. Maguire Jr.[1], and Dejan Kostić[1]

[1]KTH Royal Institute of Technology
[2]Ericsson Research

Abstract

Data centers increasingly deploy commodity servers with high-speed network interfaces to enable low-latency communication. However, achieving low latency at high data rates crucially depends on how the incoming traffic interacts with the system's caches. When packets that need to be processed in the same way are consecutive, i.e., exhibit high temporal and spatial locality, caches deliver great benefits.

In this paper, we systematically study the impact of temporal and spatial traffic locality on the performance of commodity servers equipped with high-speed network interfaces. Our results show that (*i*) the performance of a variety of widely deployed applications degrades substantially with even the slightest lack of traffic locality, and (*ii*) a traffic trace from our organization reveals poor traffic locality as networking protocols, drivers, and the underlying switching/routing fabric spread packets out in time (reducing locality). To address these issues, we built Reframer, a software solution that deliberately delays packets and reorders them to increase traffic locality. Despite introducing μs-scale delays of some packets, we show that Reframer increases the throughput of a network service chain by up to 84% and reduces the flow completion time of a web server by 11% while improving its throughput by 20%.

1 Introduction

Recent advances in networking hardware have boosted the speed of Network Interface Cards (NICs) and packet switching devices, facilitating faster Internet access [1, 2] and improving performance in datacenters [3]. At the same time, this sudden growth in networking speeds has not been followed by a similar trend in Central Processing Unit (CPU) core frequencies and memory access latencies [4, 5]. This places tremendous pressure on today's commodity server architectures. Accessing main memory for each packet is prohibitive, thus high-speed packet processing inherently requires packets *and* the instructions & data needed to

process these packets to reside in cache memories to the greatest extent possible. For these reasons, recent efforts have explored ways to optimize cache utilization, for instance, (*i*) using Direct-Memory Access (DMA) or Remote DMA (RDMA) [6] to eliminating CPU involvement in the reception of incoming packets, (*ii*) with Data Direct I/O (DDIO) [7, 8] completely avoiding main memory, (*iii*) placing incoming packets into a Last Level Cache (LLC) slice as close as possible to the core responsible for handling these packets [9], and (*iv*) realizing Network Function (NF) chains *without* inter-core communication (thus eliminating LLC cache pollution) [10] and *with* whole-stack optimizations (minimizing LLC accesses) [11].

Optimal utilization of memory caches requires that packets to be processed (with a given set of instructions *and* data) arrive as close as possible in time to each other, i.e., high *temporal* and *spatial* locality of the received packet stream. In this paper, we investigate the impact of packet ordering on the performance of I/O-intensive applications. We first measure a variety of performance metrics including throughput, average processing cycles per packet, average CPU instructions per packet, etc., as functions of the level of traffic locality of a set of streams of packets . In our experiments, the relevant data is both packets belonging to *the same flow* and the metadata that is associated with them. Our investigation reveals an unexpected sharp performance degradation (up to a factor of 3×) with even the slightest lack of temporal and spatial traffic locality for packets that could have been processed using the same instructions and data. As an example, we discovered that the number of CPU cycles per packet for an iperf server were reduced by a factor of 2−3× when packets arrive in small bursts of 5 packets belonging to the same flow as opposed to bursts of a single packet.

In practice, there are several hindrances to cache-optimized I/O processing. First, slow NICs at the client do not produce bursts of packets that will arrive "back-to-back" at a receiver with a faster NIC. Moreover, the multiplexing of different traffic flows along the path from a client to a server results in packets belonging to a client's flow being spaced apart

(i.e., interleaved with other flows), thus diminishing locality. Even worse, we observe the existence of an increasing *friction* between emerging networking trends, which advocate that congestion control mechanisms pace packets, i.e., spread packets in a flow apart from each other as much as possible to minimize the risk of congestion in the network (see §3), and the desire to process incoming packets in memory caches to the greatest extent possible (due to trends in computer architecture) [12]. To understand whether real-world traffic exhibits sufficient ordering, we analyzed a real-world traffic trace from one of the packet gateway interfaces of our organization. This traffic exhibits a very low level of spatial locality, as more than ~60% of the packets belonging to the same flow are interleaved with packets belonging to other flows, which is far from ideal conditions for cache-optimized packet processing.

These apparently completely opposite requirements of (*i*) pacing traffic for better network-level statistical multiplexing and (*ii*) processing packets in bursts for better cache effectiveness calls for a solution that satisfies both requirements at the same time. Based on the above, we explore the counter-intuitive idea of *increasing packet processing throughput by deliberately delaying and reordering packets before they reach the application running on the server(s)*, thus rebuilding high traffic locality. We built Reframer, a network function that leverages this idea, to buffer and reorder packets between different flows. By introducing Reframer at the *destination* network (or directly at an end server), we (*i*) maximize the number of subsequent cache hits in the servers, thus reducing the processing time for each burst and (*ii*) are compatible with the emerging pacing-based congestion control mechanisms (e.g., BBR [13]) as we do not affect the pacing of the packets across the Internet. Reframer can be deployed on the same server where one needs to increase cache hit performance (e.g., CPU core and/or SmartNIC) or upfront as part of a network function service chain to improve the throughput of the service chain itself by up to 60% (see §5.2) and subsequent web servers throughput by 20% while reducing the flow completion time by 11%, despite delaying the individual packets. Moreover, we show that Reframer improve performance an order of magnitude more than flow-oblivious batching [14], showing the need to increase *per-flow* spatial locality.

Contributions. In this paper, we:

- Unveil that trends in networking, spreading packets apart, are antithetical to today's high-performance computer architectures, which require bursty communication to efficiently use cache memories for high-speed networking.
- Systematically measured the performance degradation due to the lack of spatial locality in the streams of packets processed by servers for a variety of I/O-intensive applications (including large data transfers and network functions). Our results show significant performance degradation, up to a

factor of $2-3\times$, mainly due to cache misses (§2).
- Analyzed the levels of spatial and temporal locality in real-world traffic captured between our organization and our ISP. This traffic shows poor locality, which leads to sub-optimal performance at each of the servers (§3).
- Built a Reframer prototype to reorder packets, thus exploit servers' caches when processing packets at high speed (§4). Reframer improves the throughput and latency of chained NFs by up to 84% and 46% respectively, using a realistic packet trace and various Reframer deployments (§5).

2   How Much Does Order Matter?

This section shows how explicit packet ordering increases temporal and spatial locality and, consequently, boosts the performance of real-world applications. Our results show that, when packets belonging to the same flow are interleaved by even a few other packets, the latency of a packet processing application may increase by more than $2\times$ because of a higher number of cache misses and executed CPU instructions. These results motivate our Reframer system, whose goal is to build *per-flow* batches of packets that can be submitted to the servers, as opposed to batches of *arbitrary* packets belonging to *different flows* as in state-of-the-art software switches (e.g., Batchy [14]).

The experimental methodology used in this section is described in §2.1. We decompose the effects of packet ordering into three categories: network stack effects (§2.2), software switching effects(§2.3), and more advanced NF effects (§2.4).

2.1   Experimental Setup

Testbed. All the experiments in this section use the same testbed. Two back-to-back interconnected servers, each with a single-socket 8-core Intel® Xeon® Gold 5217 (Cascade Lake) CPU clocked at 2.3 GHz and 48 GB of DDR4 RAM at 2666 MHz. Each core has 2×32 KiB L1 (instruction & data caches) and 1 MiB L2 caches, while one 11 MiB LLC is shared among the cores. Each server has a dual port 100 GbE Mellanox ConnectX-5 NIC with firmware version 16.28.1002. Hyper-threading is enabled on both servers and the Operating System (OS) is the Ubuntu 18.04 distribution with Linux kernel v5.3. One server acts as a traffic generator and receiver while the other server is the Device Under Test (DUT). We also utilized the Linux `perf` tool on the DUT during the execution of the experiments to monitor CPU performance counters (e.g., CPU cache misses).

Spatial locality factor (*SLF*). We define *SLF* as the average number of packets, in the same flow, that arrive back-to-back at the DUT. For example, if there are three flows (A, B, and C) and $SLF = 1$, the DUT receives packets in the pattern "ABCABC...". For $SLF = 2$, the pattern is "AABBCC...".
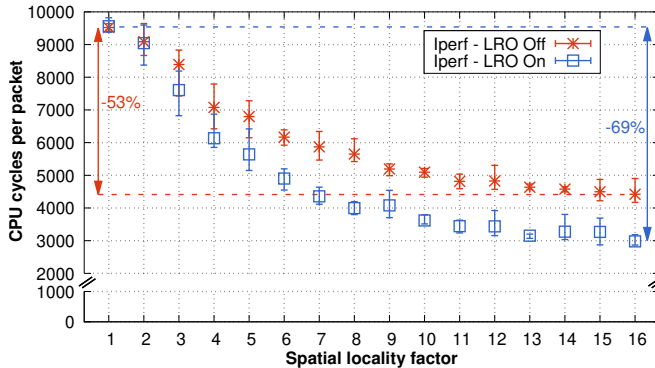
(a) CPU cycles per packet.
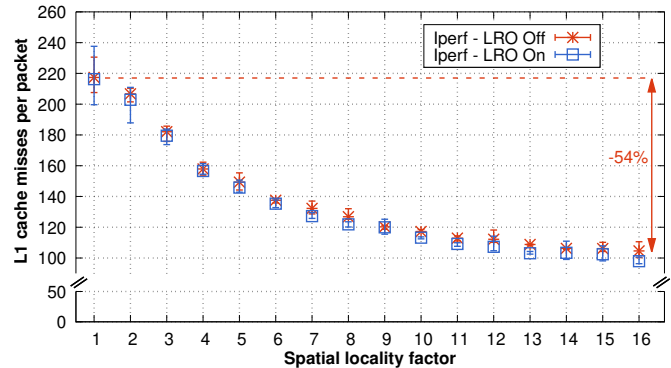
(b) L1 cache misses per packet.

Figure 1: Impact of packet spatial locality on the performance of an `iperf` server, with and without LRO.

2.2 Network Stack Effects

Packet ordering has a profound impact on the performance of general purpose network stacks and their applications, especially TCP receive-side processing. In these experiments, we show that lack of traffic locality greatly degrades CPU utilization (up to a factor of $3\times$) even when sophisticated TCP accelerations are used.

In these experiments, we use Linux `iperf` [15] to establish 128 TCP connections (with 1500 B packets) to the DUT that runs an `iperf` server. The duration of each test is 15 seconds.

We utilize the Linux traffic control mechanism (`tc`) on the client side to synthetically order the sending packets with a given value of *SLF*. We restrict the sending rate to ~8 Gbps, as forcing a real TCP stack to exhibit a specific *SLF* at high speeds is extremely hard. On the DUT side, we restrict `iperf` to use only one core to clearly delimit the benefits of packet ordering from potential artefacts introduced by parallelism.

Lack of locality makes TCP accelerations ineffective. A variety of TCP accelerations have been devised to mitigate the effects of the increasingly faster NICs' transmission speeds on the relatively stable CPU speed. In this experiment, we show that the most notable of these accelerations, i.e., Large Receive Offload (LRO), is ineffective with low traffic locality.

Ideally, LRO should combine *SLF* consecutive packets of the same flow received at the NIC into a single "super-frame", removing all the Ethernet & IP headers from the merged packets and possibly coalescing redundant packets, such as TCP acknowledgements. However, interleaved packets from different flows prevent LRO from merging consecutive packets which leads to inefficiency of LRO.

The blue boxes of Fig. 1a show that LRO performance is improved significantly when the spatial locality factor increases from 1 to 16, i.e., more consecutive packets in a flow arrive at the DUT. This increase in *SLF* reduces the number of CPU cycles per packet by 69% (from ~10k to ~3k), which shows low traffic locality harms TCP acceleration by LRO. Even without LRO (red boxes in Fig. 1a), the number of CPU cycles per packet decreases by 53% with an increasing *SLF*.

Two explanations for the benefits of spatial locality are:

① **Fewer cache misses.** Ordered packets increase L1 cache hit ratio as common per-flow data structures are fetched only once for all packets. Fig. 1b shows that the number of L1 cache misses per packet decreases by 54% when packets are processed back-to-back. Particularly, we observed an increase in performance for the "__inet_lookup_established" Linux kernel routine. This function performs a lookup in the listening sockets hash table to assign the received packet to the corresponding socket. The improvement is identical regardless of whether LRO is enabled or not and simply depends on having a better packet locality.

② **Fewer CPU instructions per packet.** Since `iperf` uses multiple threads to serve clients' requests, when *SLF* is small, the scheduling routines of the Linux kernel are called more frequently to switch among `iperf` threads. By increasing *SLF*, each thread is able to handle multiple consecutive packets (ideally *SLF* packets) within a single scheduling round of the Linux kernel. Consequently, the number of flow handling routines and executed CPU instructions decreases dramatically with or without LRO enabled (see Fig. 2). LRO further reduces the average number of CPU instructions per packet thanks to the creation of super-frames of packets.
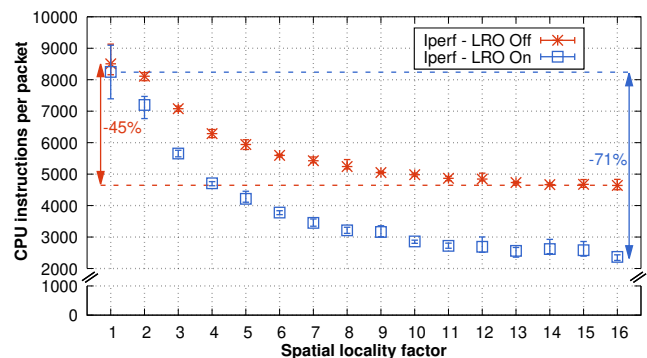

Figure 2: Impact of packet spatial locality on CPU instructions per packet of an `iperf` server, with or without LRO.
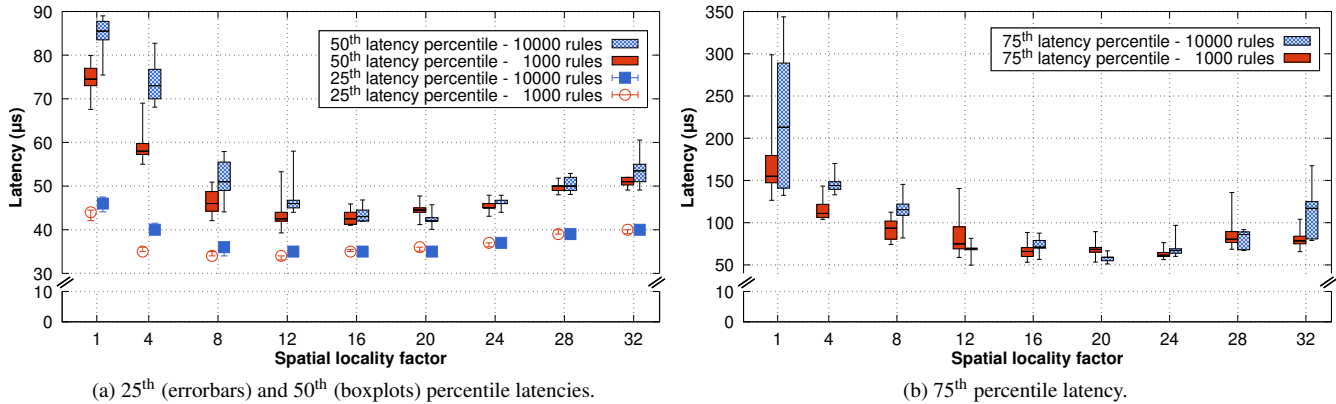
(a) 25th (errorbars) and 50th (boxplots) percentile latencies.

(b) 75th percentile latency.

Figure 3: Impact of spatial locality on the forwarding performance of OVS 2.13.9 using the Linux kernel v5.3 data path.

Takeaway. From this experiment we conclude that the performance of today's high-speed networking applications is highly dependant on the spatial locality of the received packets, as this impacts cache-miss ratios and the number of CPU instructions per packet. Based on Fig. 1a, we observe that systems without LRO acceleration but with good spatial locality of packets (i.e., $SLF = 16$) perform *better* than systems with LRO but with poor locality of packets (i.e., $SLF < 5$), making it beneficial to process ordered streams of packets.

2.3 Software Switching Effects

This section quantifies the effects of locality on the performance of the kernel-based Open vSwitch (OVS) [16]; a widely deployed production quality multi-layer software switch. Many Virtual Machine (VM) and container-based cloud platforms (e.g., VMware NSX-T [17], OpenStack [18], Red Hat's OpenShift [19], and Kubernetes [20]) use OVS.

OVS classification pipeline. Upon a packet's arrival, OVS employs a multi-stage classification pipeline. The first stage is a 2^{13} entry Exact Match Cache (EMC) for frequently used flows. This cache uses a 32-bit hash of the packet's header, which can be the Receive-Side Scaling (RSS) hash, as a key mapped to a rule for the corresponding packet. In OVS 2.10, a second classification stage called Signature Match Cache (SMC) was introduced as an experimental feature. This cache stores a 16-bit signature for each flow along with a corresponding 16-bit index into a flow table (with up to 2^{16} rules), a total of 32 bits; hence, it is more memory efficient than EMC, which stores the entire forwarding rule.

If neither of the first two cache levels matches an incoming packet, then that packet is classified by the kernel's Megaflow cache [21]. This cache is based on the Tuple-Space Search (TSS) algorithm [22] that uses more aggressive bitwise wildcarding to aggregate multiple flows into a single match. Finally, a miss in the Megaflow cache results in a packet redirection to the "slow path", where packets traverse a pipeline of OpenFlow tables to derive their corresponding actions.

OVS setup choice. Due to the fact that the EMC is an n-way associative cache (similar to a modern CPU cache), only n out of 2^{13} entries can be used to store any given flow. In OVS version 2.13.9 $n = 2$, implying this cache will likely exhibit high contention even when the number of flows is much smaller than the EMC. Measurements of these OVS caching schemes showed that EMC does not yield the expected levels of performance improvements over the SMC [23]. Specifically, EMC slightly outperforms the SMC *only* with low numbers of flows (< 200), while SMC offers higher performance with more flows [23]. We verified this through our own experiments, hence we disable EMC to achieve higher performance.

OVS experiment. We deployed OVS 2.13.9 on the DUT with a data path through the Linux kernel v5.3 of the DUT. The forwarding behavior is defined by two sets of OpenFlow v1.4 rules with 1k and 10k entries. These rules classify input packets based on their source and destination Ethernet and IP addresses and forward matching packets toward the traffic receiver through the same port (i.e., the Mellanox port of the DUT attached to OVS). Only one rule in each rule set matches the input traffic. We used a Data Plane Development Kit (DPDK)-based traffic generator to inject a trace of 10k User Datagram Protocol (UDP) flows, where each flow consists of 1500-B packets, at the rate of 5.5 Mpps $\approx$ 66 Gbps*. Fig. 3 shows the performance of the kernel-based OVS classifier, focusing on the 25th & 50th (Fig. 3a) and 75th (Fig. 3b) latency percentiles.

Packet ordering greatly benefits OVS's caching scheme. When no particular locality is enforced (i.e., $SLF = 1$), the 75th latency percentile (see Fig. 3b) ranges between 132 μs-343 μs and 126 μs-300 μs for 10k and 1k rules, respectively; while lower latency variance is observed in Fig. 3a for the 25th and 50th latency percentiles. However, both latency and its variance substantially decrease with increasing SLF for both rule sets. The greatest improvement is observed for $SLF \in [20, 24]$, where packet locality results in $2.5 - 5\times$

---

*Similar results occur for TCP packets. With 64-B packets, the effect of packet ordering is less profound, but still relevant.

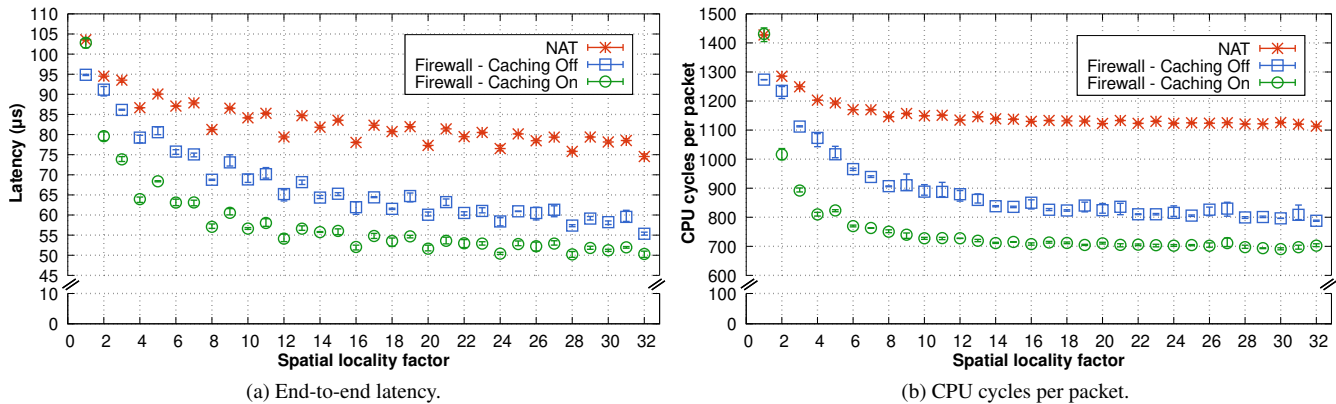(a) End-to-end latency.   (b) CPU cycles per packet.

Figure 4: Impact of traffic spatial locality on the packet processing latency and the CPU cycles per packet performance of a NAT and a firewall (with and without rule caching) NFs.

lower 75th latency percentiles, 2× lower medians, and 15-22% lower 25th latency percentiles with negligible latency variance. For higher values of the spatial locality factor (i.e., $SLF \in [28, 32]$), we observe a slight latency increase compared to the lowest attainable latencies shown in this figure. This behavior is not observed in the other experiments in this section, suggesting the limits of packet ordering be studied on a case-by-case basis.

10k rules at the cost of 1k rules. An equally important benefit of this use case is shown in the case of $SLF \in [20, 24]$, where the red and blue boxplots and error bars in Fig. 3 exhibit very similar ranges. This means that packet ordering amortizes the additional cost of a 10x larger classifier (i.e., 10k vs. 1k rules) by making the most out of OVS's caches.

2.4 Network Functions' Effects

In addition to network stacks (§2.2), packet locality may also affect more advanced NFs. To investigate this, we implemented two NFs in FastClick [24], a stateless firewall and a (stateful) Network Address Translation (NAT)*. Unlike §2.2, we allocate two cores per NF with one RX queue per core to show that the benefits of packet locality is not limited to single-core scenarios. We will further discuss the impact of number of RX queues on the DUT performance in §5.1. In these experiments, the traffic generator emulates 10k clients sending a total of 20 million 1-KB UDP packets to the DUT with a total rate of ~50 Gbps (6.2 Mpps) and a given spatial locality factor SLF. Fig. 4 shows the average end-to-end latency and the number of CPU cycles per packet for these two applications.

NAT NF case. We deployed the NAT NF on the DUT. Fig. 4 shows that the end-to-end latency decreases from 103 μs to 74 μs as the spatial locality factor increases from $SLF = 1$ to $SLF = 32$. When $SLF = 1$, some packets are dropped since for each packet, the CPU must wait for the many cycles

it takes to fetch the appropriate NAT table's row from the memory, greatly decreasing the available useful processing time and the capacity of the NF to serve incoming packets. In contrast, when input packets are partially ordered by flow, the NF amortizes the cost of this NAT table lookup over several consecutive packets within the same flow, thus reducing the average processing time needed to serve each packet.

Firewall NF case (without software-based rule caching). We deployed a firewall NF implementing a tree-based Access Control List (ACL) with 20k rules on the DUT. We consider two different variants of this firewall. The first variant assumes *no* rule caching, thus it executes the matching algorithm for each incoming packet. Since all packets of the same flow typically match the same rule, then with an increasing spatial locality factor, we expect a reduction in the frequency of fetching data (rules) from main memory into the system's cache(s). The blue boxes in Fig. 4 show similar trends as in the previous experiment, i.e., an increasing spatial locality factor improves the performance of the firewall in terms of both average end-to-end latency (Fig. 4a) and number of CPU cycles per packet (Fig. 4b).

Firewall NF case (with software-based rule caching). The second variant of this firewall NF implements a simple in-memory rule cache. This cache stores the hash of the last served packet and the matched rule. For each incoming packet, the firewall calculates the packet's hash value, and if it is the same as the entry in the cache, then it assumes that the packet will match the same rule as the previous packet. However, if after executing the rule the new packet does not match the rule, then the cache will be updated with a new matching rule and a new packet hash. The green circles in Fig. 4 show faster convergence to the minimum values compared to the firewall *without* caching as the firewall's cache matches an increasingly large fraction of input packets (i.e., $SLF - 1$ packets for a given SLF) without invoking the firewall's classifier.

Packet spatial locality analysis. Looking closely at the per-packet CPU cycle curves shown in Fig. 4b, we note that the

---

*We also deployed a chain of NFs on the DUT as a complementary experiment in Appendix A.1
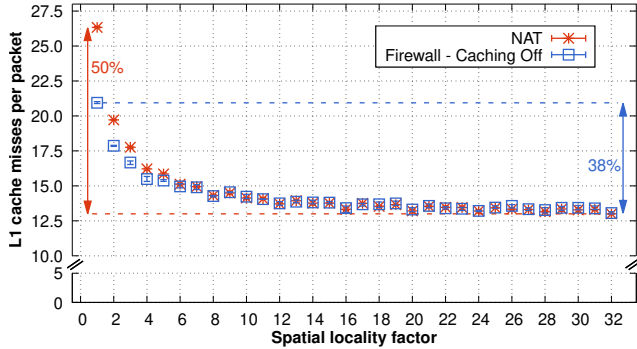
Figure 5: Impact of spatial locality on the number of L1 misses per packet for a Firewall (w/o caching) and NAT NF.

data fits an equation of the form $cost = \alpha * (1/SLF) + \beta$, where β is the CPU cost of processing the data that has already been accessed and is in the cache, hence it is the asymptotic limit when SLF is large. In contrast, $\alpha * (1/SLF)$ is a weighted version of the cost of getting the data that can be shared, e.g., when $SLF = 2$, the cost per packet is amortized over two packets.

In the case of the NAT, when $SLF > 1$ the main cost is the lookup of the appropriate replacement values in the NAT table and this lookup only has to be done once for the first packet, hence $\alpha \approx 1$ times the cost of this lookup. In the case of the firewall, we expect that for a given number of firewall rules F, $\beta \propto \gamma * F$ when the firewall rules cannot be cached (i.e., when the rules cannot fit into the cache), hence the firewall rules have to be repeatedly loaded and hence the cost cannot be shared (i.e., $\alpha \approx 0$). However, we see that this is not the case in Fig. 4, as an application still benefits from processor-based caching of the data even *without* software-based rule caching.

Serving packets at the speed of L1 cache. We now highlight the fundamental role played by core-specific L1 cache in enhancing the performance of the above NFs. To measure cache-related events, we utilized the Linux `perf` tool during the execution of the experiments shown in §2.4. Since the NFs' data size (NAT table and firewall rules) are smaller than the LLC and L2 capacity, we see almost no LLC and L2 misses; hence, the reduction in the number of CPU cycles is mostly due to better utilization of the L1 cache.

Fig. 5 shows the effect of locality on the number of L1 cache misses for both the NAT and firewall experiments. In both cases, we observe a substantial decrease in the number of L1 cache misses. Our analysis reveals that we can observe the effects of ordering even on the L2 and LLC misses by deploying a memory-intensive NF (e.g., Deep Packet Inspection (DPI)) or a chain of multiple NFs on the DUT (Appendix A.1). Our results demonstrate that better utilization of core-specific caches is the key for increasing the NFs' performance *and* ordering packets minimizes cache misses.

2.5 Summary

In this section, we explored the effects of spatial locality of network data by conducting experiments across Linux network stack and DPDK-based stateless & stateful NFs at various levels of a system's software stack. The common denominator of this study is that packet ordering greatly increases the utilization of a server's memory hierarchy (mostly CPU caches), which in turn results a substantial improvement in key performance indicators, such as latency, throughput, and CPU utilization.

We leverage these insights to design a system that vertically (i.e., hardware to application layer) exploits the benefits of packet ordering (see §4) and demonstrate complementary results using additional real world applications (see §5). Before this, we investigate whether today's Internet traffic exhibits a low or high spatial locality factor (see §3).

3 Packets Order in Real-world Traffic

This section analyzes a trace from our organization (i.e., a university) to understand the spatial & temporal locality in realistic traffic (§3.1) and explores opportunities to increase traffic locality by reordering packets (§3.2). Our analysis shows that >60% of the packets belonging to a flow are interleaved with packets of other flows, hence non-ideal for high-speed packet processing (based on §2). Moreover, today's networking trends further exacerbate this – as novel congestion control mechanisms (e.g., BBR [25], Timely [26], HULL [27], and Carousel [28]) advocate pacing packets to fight "bufferbloat", i.e., keeping queue occupancy in routers' buffers as low as possible. Even the built-in self-clocking of traditional TCP congestion control mechanisms [29], which inherently spreads packets out over time to avoid congesting a link, is harmful to cache-optimized high-speed network communication. In §4, we advocate rebuilding per-flow traffic bursts as close as possible to the servers that process them.

Trace statistics. We captured 28 min of traffic from our campus to & from our upstream network provider. The outgoing traffic (i.e., from the campus toward the Internet) had 420 million packets with an average size of 1069.43 B and the incoming traffic (i.e., from the Internet toward the campus) had 378 million packets with an average size of 882.82 B. Fig. 6 shows the TCP flow size distribution for this traffic. In the rest of this document, we refer to the outgoing and incoming traffic as the TX and RX traces, respectively.

3.1 Spatial & Temporal Distance

The performance benefits of packet spatial locality were shown in §2 with the greater the number of consecutive packets belonging to the same flow (i.e., the spatial locality factor), the greater the benefits. Additionally, we concluded that even a small spatial locality factor (e.g., $SLF = 5$) could yield a
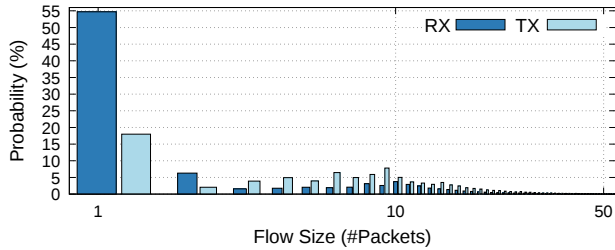
Figure 6: TCP flow size distr. of the analyzed trace with a log. x-axis. The RX trace has $\sim$ 4M flows; the min., avg., and max. flow sizes (in #packets) are 1, 63, and $\sim$ 29M, resp. The TX trace is composed of $\sim$ 2M flows; the min., avg., and max. flow sizes (in #packets) are 1, 137, and $\sim$ 68M, resp.

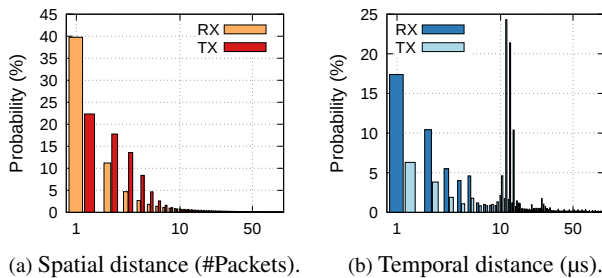(a) Spatial distance (#Packets). (b) Temporal distance (μs).

Figure 7: Distribution of the spatial & temporal distance for the campus trace. (Note that the x-axis is logarithmic).

significant improvement, as was shown in Fig. 4. However, the improvements depend on the traffic's actual spatial locality. Therefore, in this section, we examine how *unordered* the trace from our organization is. To do so, we calculate the spatial and temporal distance of packets in every TCP flow. *Spatial distance* shows the number of packets between two consecutive packets of the same flow and can be used to assess opportunities to exploit cache memories. The higher the spatial locality, the greater the number of opportunities to increase cache-hit ratios. *Temporal distance* measures the time between two consecutive packets of the same flow and can be used to estimate how long one would have to wait for another packet in order to reorder packets and increase spatial locality. Fig. 7 shows the histogram of these metrics for the campus trace. These results do not consider single-packet flows*, as these metrics are undefined for such flows.

Spatial distance. Fig. 7a shows that the spatial distance of the per-flow packets are larger than one packet in ~60% of the RX trace (without single-packet flows) and ~75% of the TX trace (without single-packet flows) – i.e., there is *at least* one packet between consecutive packets of the same flow. The rate of our campus trace is $\sim$2.2 Gbps, which underestimates the values reported for the spatial distance. In networks with

---

*Based upon the source addresses, we expect that some of these are likely to be part of SYN attacks.
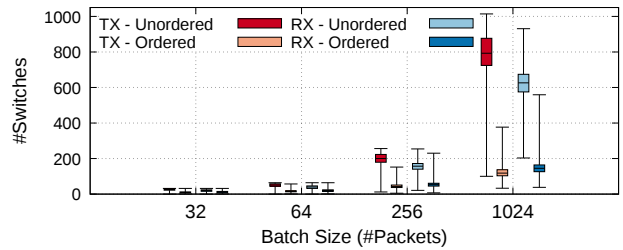
Figure 8: Number of per-flow switches for different batch sizes (selected according to [14]): 32/64 for Linux kernel and DPDK; 256 for VPP; and 1024 for GPU/NIC offload).

higher rates (e.g., multi-tens- & multi-hundred-gigabit rates), the spatial distance would *intuitively* be much larger, which further reduces the locality. As shown in §2, this lack of spatial locality can dramatically degrade performance, up to factor of $3\times$. Fig. 8 shows the number of switches across different flows that an application should theoretically perform when processing different batch sizes of packets. The number of switches can be more than $5\times$ larger when the packets are unordered. Frequent switching could cause detrimental performance events (e.g., context switches and/or cache evictions), the number of which depends on the system's microarchitecture (e.g., cache hierarchy) and the application characteristics (e.g., the type of processing and the size of the per-flow state).

Temporal distance. Fig. 7b demonstrates that temporal distance between consecutive flow packets in a flow is typically smaller than a few tens of microseconds, making it possible to reduce the spatial distance by buffering packets for a short time so that they can be reordered. The potential for reordering of traffic destined/originated to/from two cloud providers is described next.

3.2 Potential of Per-flow Ordering

We identified the top hundred IP addresses of the TCP connections, which appeared in independent flows of the TX trace. From those, we select those of two popular cloud providers, referred to as $Cloud_1$ and $Cloud_2$[†]. We calculated the probability of receiving packets of the same TCP flow within different fixed-size time windows to determine whether by waiting for a short amount of time we can reorder packets to make *per-flow* batches of packets, i.e., regenerate high spatial locality. Additionally, since user-space packet processing frameworks (e.g., DPDK) use a fixed batch size for processing packets (typically 32 for a DPDK-based application), we assume that up to 32 packets per flow can be buffered[‡]. Fig. 9 shows the distribution of batch sizes for different buffering times. These results consider all flow sizes, including single-packet and mice flows which dramatically reduces the size

---

[†]Table 1 (in Appendix A) shows the statistics of these flows.
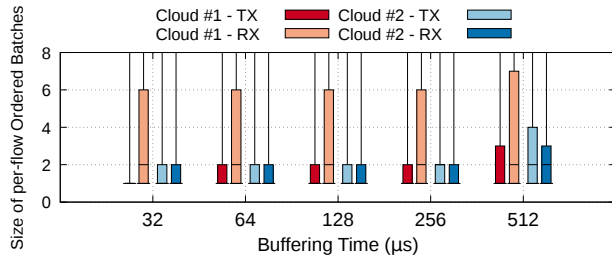[‡]In some cases it might be possible to buffer up to ~300 packets, see Fig. 23 (in Appendix A).

Figure 9: Impact of increasing the waiting time on the probability of receiving packets in the *same TCP flow* (i.e., packets going to the same end-host, the same core, and the same application).

of the per-flow batches. Clearly, increased buffering time is positively correlated with receiving more packets in the same flow. However, the statistical properties of traffic (e.g., a cloud service) should be taken into account when ordering packets. For instance, it is possible to make per-flow batches of size 6, even in 32-μs time frames, for 25% of the incoming traffic from $Cloud_1$; whereas 25% of the outgoing traffic toward $Cloud_2$ could only be made into per-flow batches of size 4.

Summary. This section showed that *most* of the flows in a campus trace could benefit to some extent from ordering, as it increases locality, i.e., decreases both spatial and temporal distances. However, the improvements depend on the traffic characteristics and type of service. Ordering of larger flows can potentially lead to much bigger improvements (see the tails of the box plots in Fig. 9). Therefore, a cloud provider/operator might only apply reordering to specific services and/or tune the waiting time based on the Service Level Objective (SLO) and the flow rate.

4 The Reframer Design

As we have shown in Section 2, receiving unordered packets leads to high cache misses and more CPU cycles per packet, which increases the cost of packet processing in networking devices. This section presents our proposal to achieve increased end-to-end data locality (both temporal and spatial) of packets in each flow. Our solution maximizes locality and is compatible with today's trends in Internet congestion control paradigms that pace packets. We leverage the idea of briefly buffering, delaying, and reordering the (possibly paced) incoming packets to increase spatial locality for network traffic. As a result, Reframer pays the imposed price of receiving paced packets only one time at the beginning of a NFs and applications chain instead of allowing every single NF pays that for itself. Reframer is developed as a *software* solution that uses CPU cycles to classify flows and create batches with higher locality. Following the trend of Network Functions Virtualization (NFV), advantages of software network functions like Reframer include more flexibility, faster development cycles, and nearly no resource limitation

(e.g., number of per-flow queues) in comparison to hardware alternatives. On the other hand, similar to many networking software systems, the main design challenge is efficiency in terms of both time and space complexity: one needs to strike a delicate balance between the complexity of the reordering procedure, which *consumes* CPU cycles, and the gains at the application/NFs, which *saves* CPU cycles; Hence, it is crucial for Reframer to employ an optimized data structure that takes a short time and space for reordering packets regardless of incoming packets rate and the number of concurrent flows. With Reframer, the incoming packets are efficiently buffered and reordered and then delivered to their destinations. Fig.10 shows the operation of Reframer when a stream of packets belonging to three flows (i.e., green, blue, and brown) arrive at the Reframer.

Flow classification. Reframer maintains two main data structures to reorder packets: a *flow classification table* and a *flush list*. For each flow, the flow table stores the timestamp (TS) when the first packet of that flow has been added to the batch of that flow. It also stores a pointer to the list of the buffered packets for that flow. The flush list is a double-linked list that stores flow identifiers sorted by timestamp described in the flow table. Reframer updates all these data structures in constant time for a variety of operations: buffering of a packet in the flow table (when a flow entry already exists), adding/removing flow identifiers to/from the flush list, finding the oldest flow identifier, and emitting a batch of packets. Only insertion of new flows in the flow table is not performed in constant time because of the cuckoo-hash table. Additionally, Reframer stores only a few bytes of metadata per flow that allows CPU cores to work at the speed of L1 and L2 caches.

In case the number of packets in the flush list meets a configurable limitation (*maximum burst size*), Reframer passes the batch to the scheduler.

Buffering Time. The flush list can buffer flows for a maximum amount of time, which we call the *buffering time* (T_{buff}). The optimal buffering time mostly depends on two parameters: (*i*) flows' average throughput and (*ii*) the end-to-end latency between a Reframer instance and the destination. The former parameter affects the possibility of receiving multiple packets of the same flow in a short time window. For instance, the inter-arrival time of 1000 B packets is 8 μs at 1 Gbps and Reframer can rebuild a per-flow batch with up to 8 packets by buffering packets for 64 μs. The latter parameter sets the upper bound for the buffering time., i.e., a higher end-to-end latency provides more flexibility to wait for packets. It is possible to adjust buffering time by automatically calculating both of these parameters; However, in the current version of Reframer, it should be configured manually by an operator.

Reframer collects additional information to track its efficiency, i.e., (i) it measures the amount of time that flows were being delayed without actually receiving more packets, and (ii) it records the average amount of packets per batches. These statistics could potentially enable Reframer to fine-tune
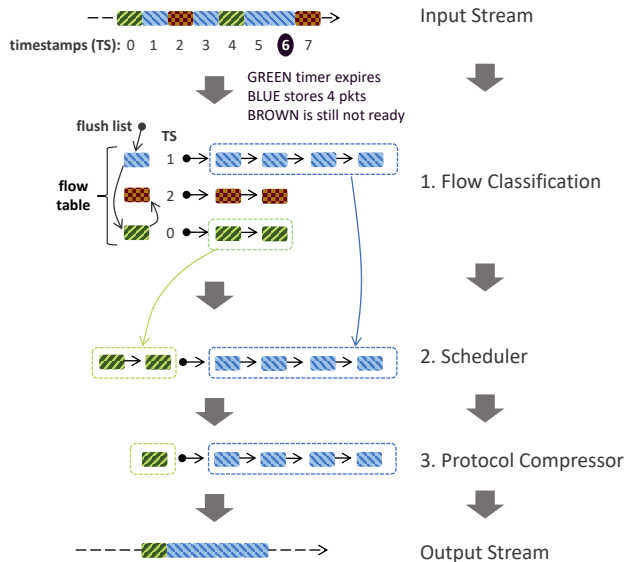
Figure 10: Reframer consists of 3 components: (*i*) a classifier arranges input packets to a flow table, (*ii*) a scheduler flushes flows from the table upon a timeout or burst-size, (*iii*) a compression module coalesces packets to eliminate redundancy.

the buffering time by finding the sweet spot between these two statistics for each application.

Priority. When either the flow classifier informs the scheduler of a full-size batch that is ready to be forwarded or the flush list contains flows reaching the buffering time, the scheduler computes an ordering of the batches based on some configurable *priorities*.

The oldest batches in the flow table are extracted from the head of the flush list. The scheduler resets the per-flow data entries upon emission of the corresponding batches.

In the example shown in Fig. 10, we assume the maximum batch size is 4 packets and the maximum buffer time is equal to 6 time units. At time $t = 7$, Reframer receives the fourth packet of the blue flow and at the same time the buffering time of the green flow expires since the first packet was received at time 0. Both batches are handled by the scheduler for transmission. Reframer's scheduler supports a variety of priority models for ordering batches ready to be sent:

Shortest flow first prioritizes mice over elephant flows.
Oldest flow first prioritizes older over newer flows with respect to the timestamp of the first packet; and
Oldest flow in the queue first prioritizes older over newer flows with respect to their waiting time in the queue.
We envision a tailor-made priority model based upon the network operator's SLOs.

Compressor & Output. Before leaving the Reframer, each batch of packets of the same flow passes through a per-protocol optimizer, e.g., multiple TCP ACKs are coalesced if all packets are in order between ACKs. In the future, we will also look at payload coalescing if the MTU allows it.

Reframer supports an integrated *"bypass"* mechanism. Thus, Reframer allows an operator to define class(es) of traffic that should not be reordered by Reframer based on any given field of the packet (e.g., IP DSCP field). We implemented the obvious case of TCP SYN, as a TCP SYN will never be followed by other packets; therefore, a SYN is never delayed. Additionally, in §5.3 we will show that bypassing mice flows may increase the benefit of Reframer because the possibility of receiving multiple packets of the same mice flow in a period of T_{buff} is low and it is not worth to delay such packets. However, in this work, we have not implemented a heavy hitter detection module, which is left for future optimization and it is not discussed here.

Reframer Implementation. We use FastClick [24] to build a Reframer prototype, which enables many placement scenarios at high speed as will be shown in §5. The classification and flow-state management is handled by its MiddleClick [30] extension, thus the code is only thousand lines long[*].

5 Reframer Evaluation

This section assesses the feasibility and performance of Reframer in increasing the temporal and spatial locality of a stream of traffic by briefly buffering and reordering packets. We evaluate performance at both *per-packet* and *per-flow* granularity in two scenarios: (*i*) to improve the per-packet processing throughput of an NF service chain and (*ii*) to reduce the Flow Completion Time (FCT) of TCP traffic streams served by an HTTP web server. Our results show that the NF chain throughput can be increased by ~84% and the HTTP flow completion times be decreased by 100s of *milliseconds* by simply delaying packets by few 10s or 100s of *microseconds*. Specifically, this section answers the following key questions about the opportunities and challenges in reordering packets: (*i*) *Can Reframer increase the packet processing throughput of an NF chain by increasing the traffic locality of a real-world traffic trace (§5.1)?* (*ii*) *How do the Reframer benefits vary depending on where it is deployed (separate or same server as the application, and then on a CPU core or a SmartNIC (§5.2)?* (*iii*) *How can Reframer handle latency-sensitive traffic (§5.3)?* (*iv*) *Can Reframer reduce the flow completion time of an HTTP web server (§5.4)?*

Testbed. We use the testbed presented in §2.1, running an NF service chain of the NAT and the Firewall presented in §2.2 augmented with a router and a flow statistic NF. First, we place Reframer between the traffic source and the DUT, running on a dedicated Intel Xeon E5-2667 CPU clocked at 2.3 GHz and 128 GB of RAM at 2133 MHz. This machine has two Mellanox ConnectX-6 NICs. While this introduces the cost of a supplementary machine, it gives us an understanding of the *maximum* performance achievable when processing the analysed traffic traces.

---

[*]The source code is available at: https://github.com/hamidgh09/Reframer
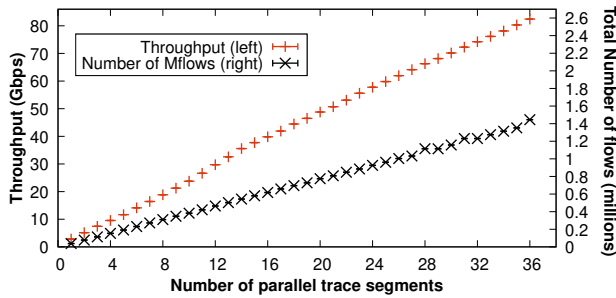
Figure 11: Traces characteristics - the X-axis is the number of multiples of our campus trace played in parallel

Workloads. We use two different types of workloads in our experiments: (*i*) *per-packet experiments on the NF chain* and (*ii*) *per-flow experiments with the HTTP web server*. The **per-packet experiments** are based on our campus traffic trace described in §3 with millions of flows in total, a throughput of ~2.2 Gbps, and an average packet size of ~1 KB. To evaluate Reframer with a higher traffic throughput, we split the traffic trace into 32 consecutive windows, each of 20 seconds, and we replay them in parallel from our traffic generator. When splitting the trace, we rewrite the flow identifiers so that any two windows do not have any flow in common (which would otherwise increase the traffic locality of the original trace). Figure 11 shows the number of flows and throughput when running a number of parallel trace segments. For the **per-flow experiment with the HTTP server**, we generate HTTP requests of 1MB files from 4096 clients using WRK [31] towards an NGINX web server.

5.1 Packet-Level Experiments (NF Chain)

In this experiment, we show that (*i*) Reframer is effective in increasing the spatial traffic locality (i.e., higher *SLF*) of our real-world traffic trace and, consequently, (*ii*) increasing the throughput of an NF chain. Since the trace is replayed, we focus on *per-packet* metrics (e.g., CPU instructions, latency) and the throughput of the NF chain. The NF chain consists of a Flow Statistic Tracker→Router→Firewall→NAT chain, all implemented in FastClick [24] using state-of-the-art NF elements and DPDK [32] for I/O. We install 10k rules into the firewall and 200 different routes into the router elements. We deploy the chain in a run-to-completion model and we consider it as the *Baseline* in all packet-level experiments. To measure the impact of Reframer, we compare the NFs chain performance *with* and *without* deploying a Reframer instance in front of the chain on an external server. Note that the latency is end-to-end in all the experiments which means it *includes* the time spent in the Reframer buffers. In this experiment, 8 CPU cores are assigned to the NFs chain with 8 RX queues on the NIC (one queue per core). The NIC uses RSS to map traffic among queues. We evaluate alternative deployments with Reframer co-located with the NF chain in §5.2.
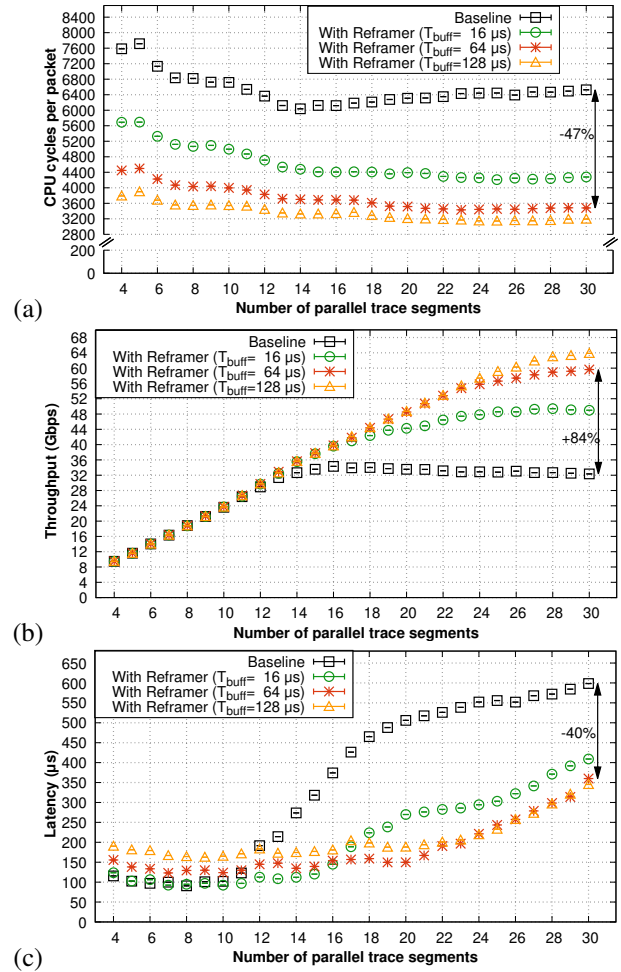
Figure 12: Performance of Reframer versus a baseline NF with increasing load when processing a real trace: (a) CPU cycles per packet, (b) Throughput, and (c) Latency.

Figure 12 shows the effectiveness of Reframer in improving the performance of the NF chain for different workloads (load is expressed as the number of parallel trace segments). At all loads, in Fig. 12(a), we see a substantial decrease in the number of CPU cycles when using Reframer. The reason is the increase in spatial locality from an average of ~1.2, i.e., near the minimum possible spatial locality, to an average of ~1.9, ~2.9, and ~3.3 at the output of the Reframer with $16\,\mu s$, $64\,\mu s$, and $128\,\mu s$ of buffering times respectively. Fig. 12(b) shows that at high loads, throughput continues to scale well for $T_{buff}= 64\,\mu s$ and $128\,\mu s$, up to ~64 Gbps (a 84-100% improvement) while the throughput peaks at ~48 Gbps for $T_{buff}=16\,\mu s$. In contrast, the baseline throughput peaks at ~33.6 Gbps and then falls - as the DUT cannot keep up. Fig. 12(c) shows that at low loads, the end-to-end latency is roughly the baseline latency plus T_{buff} when using Reframer. However, we see the Reframer benefit appears as the load increases to maximum capacity of the NFs chain. We discuss and evaluate how to reduce the additional latency introduced by Reframer in §5.3.
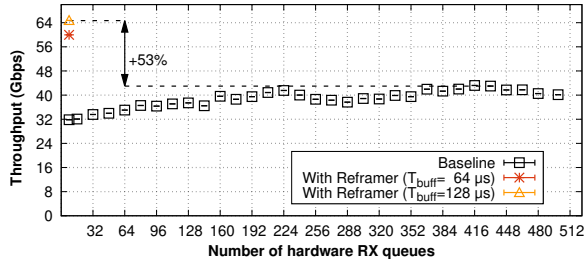
Figure 13: Reframer vs Baseline with various number of hardware RX queues (up to the max. supported by the NIC).
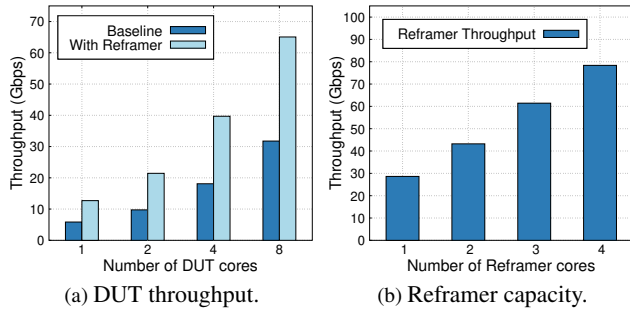

(a) DUT throughput.   (b) Reframer capacity.

Figure 14: Maximum throughput of Reframer and DUT with different number of cores.

High number of RX queues has small impact on the DUT throughput. We repeated the above experiment with 30 parallel trace segments and various numbers of RX queues on the DUT in a range of 8 to 500 (which is the maximum possible number of RX queues on the DUT's NIC). We preserve the total number of descriptors around 8192 by setting per queue descriptors to $max(32, 2^{14-\lceil \log N \rceil})$ where N is the total number of RX queues. In this experiment we show that by increasing the number of RX queues the average spatial locality increases from ~1.2 to ~2.5 without Reframer. However, despite the improvement in the traffic locality, Figure 13 shows only a slight increase in the maximum throughput of baseline. The main reason is, having hundreds of RX queues leads to more empty polling in the DUT which is costly and negatively affects the performance. It is worth noting that, fetching incoming packets from RX queues is hardware-specific and depends on the data structures that NICs are using to process incoming packets; hence, optimizing algorithms and data structures in future NICs may lead to better results. However, discussing the future road map of NICs is out of scope of this paper. On the other hand, when Reframer is located between the traffic source and DUT, increasing the number of DUT RX queues has a negative impact on the throughput because incoming packets are already sorted and classifying flows in different hardware queues does not increase packets' locality. So we set 8 RX queues (one per core) for DUT when Reframer exists in the network. Finally, we see 53% more throughput with Reframer
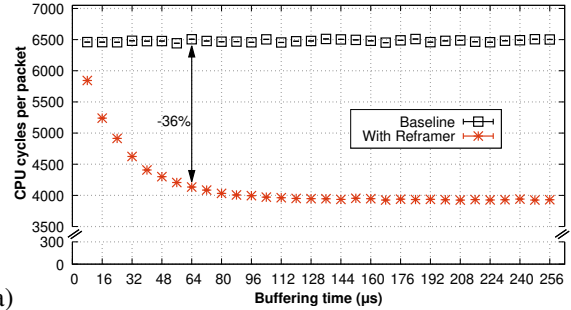
(a)
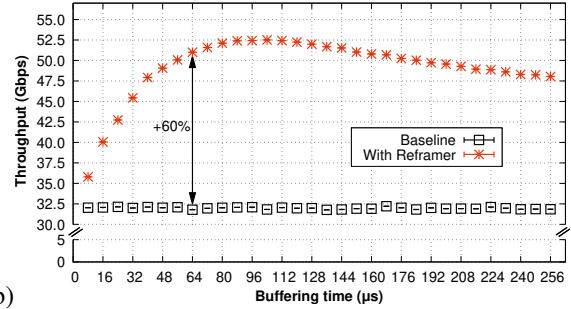
(b)

Figure 15: Impacts of Reframer when collocated with the NF chain: (a) Cycles per packet and (b) Throughput.

vs. using hundreds of RX queues for the baseline case.

Packets' locality benefit persists with various number of DUT cores. We also show that Reframer benefits do not depend on the number of DUT cores. To do so, we measured the maximum throughput of DUT by running the experiment with various number of cores assigned to DUT. Reframer's buffering time is set to 128 µs in all cases. Figure 14a demonstrates that the throughput increase rate is almost the same for different number of cores.

Reframer scales almost linearly with the number of cores. As we discussed in §4, Reframer benefits from an optimized data structure to classify, order, and flush packets in a constant time. Our stress test reveals that Reframer is able to handle up to 28 Gbps with only *one core*. Here, we increase the offered load gradually until we see ~1% packet drops in Reframer. Figure 14b shows that Reframer's capacity increases *almost linearly* when increasing the the number of cores.

5.2 Same-Server Deployment

In the previous section, we showed that deploying Reframer on a dedicated server increases spatial and temporal locality, ultimately resulting in significant performance gains. In the following experiments, we evaluate deploying Reframer on the same server where an application is running. Using the same NF chain (Baseline) as previously, we consider two deployments: (*i*) chaining Reframer with the NF chain, i.e., the entire chain running to completion on the same CPU cores (referred to as *in-chain* deployment), and (*ii*) deploying Reframer on a SmartNIC.

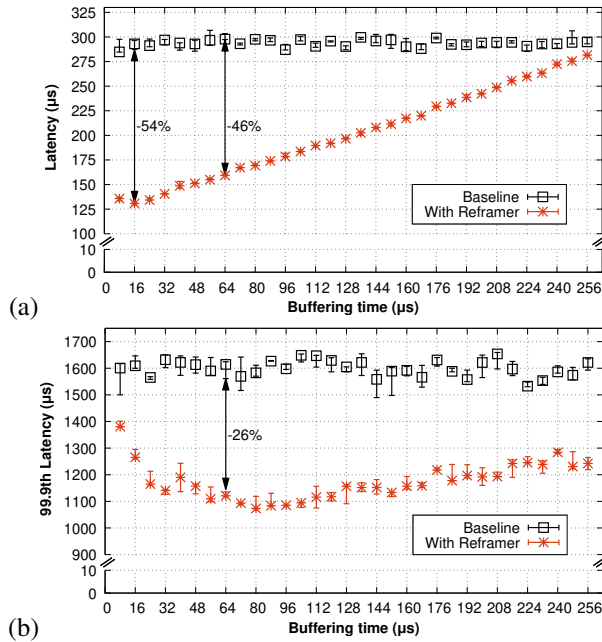**In-chain deployment.** We evaluate the performance of Re-

Figure 16: Impacts of Reframer when collocated with the NF chain: (a) Average latency and (b) 99.9th percentile latency.
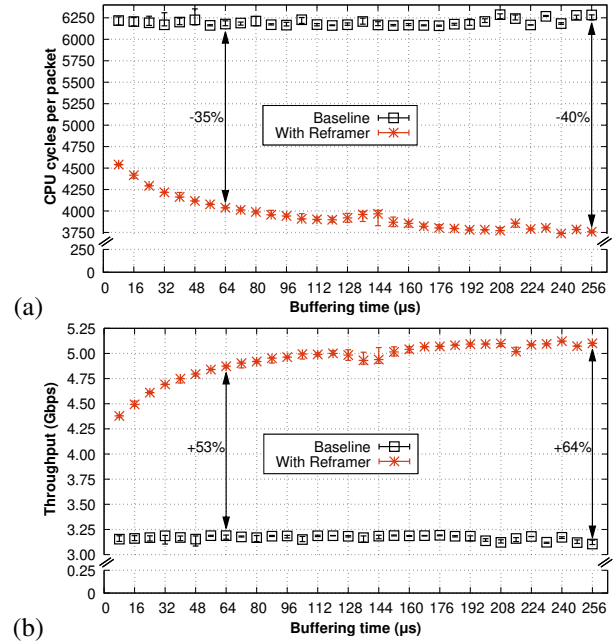

Figure 17: Impact of Reframer when offloaded into a Smart NIC which precedes the NF chain: (a) Cycles per packet, (b) Throughput. Latency is given in Appendix A.2

framer for the *in-chain* deployment versus the *baseline* for different buffering times. Generally, increasing buffering time in Reframer will lead to more packet locality, since it increases the possibility of receiving more packets of the same flow; Hence, we see a considerable increase in the DUT throughput and reduction in the end-to-end latency. Fig. 15 shows that by placing Reframer right before the service chain, the number of cycles per packet decreases with increasing buffering time while throughput increases by 60% when Reframer buffers packets for 64 μs. To evaluate the impact of Reframer on the packets end-to-end latency, we restrict the incoming packet rate to ~30 Gbps which is less than the maximum capacity of DUT in the baseline mode. In Fig. 16 we can see the average latency is reduced by 46% with T_{buff}=64 μs. Additionally, Reframer improves the tail latency by ~26% even when it is collocated with service chain on the same server. In this experiment, latency benefits start to fade gradually from a specific buffering time because the cost of delaying packets surpasses the processing speed-up. The baseline numbers are mostly the same for all x axis values because we have no buffering in baseline mode. The fluctuation in baseline values is inevitable because DUT cores are at a maximum load.

SmartNIC deployment. As a proof-of-concept deployment for offloading Reframer into a NIC to save CPU core resources on the server, we deployed Reframer on two ARM cores of a Mellanox Bluefield SmartNIC – equipped with 16×64-bit Armv8 A72 cores and two 100 Gbps ports while the NFs chain works on a single CPU core. Fig. 17 shows improvements in throughput similar to the in-chain deployment. We discovered that the performance using a single ARM core

was limited by the current Mellanox drivers for the cards, a constraint/limitation confirmed with Mellanox.

Flow-oblivious batching is highly suboptimal. We also compare Reframer with a Batchy-like [14] implementation written in FastClick. Batchy is a state-of-the-art packet processing system that buffers packets in a *flow-oblivious* manner at multiple locations in an NF chain, i.e., Batchy does not create bursts of packets from the same flow but mix all flows that must be processed by the same NF element. We observe that Batchy improves the throughput of the chained NFs by 4%, whereas Reframer improves throughput by 48%. These results corroborate our analysis in section (§2), where we showed how detrimental it is to process streams of packets that are highly interleaved between different flows as opposed to per-flow batches.

5.3 Latency-Sensitive Flows

In our previous experiments, Reframer delayed all types of packets for a T_{buff} interval, possibly increasing the FCT or packet processing time of short flows. We argue that an operator could explicitly tag which traffic classes should be delayed to improve application throughput. To evaluate the impact of delaying only large flows, we ran an experiment similar to the one described in §5.1, but we explicitly flag only large flows so that Reframer can batch them while bypassing the unflagged packets and show the results for increasing number of parallel trace segments in Fig. 18. Compared to the case where all flows are delayed, the throughput of
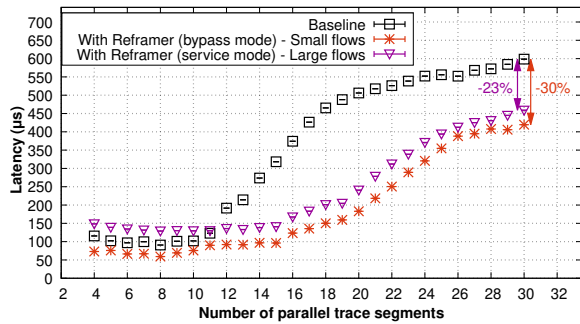
Figure 18: Reframer provides differentiated services by prioritizing small flows over large flows which are bypassed.

the NF chain slightly decreases (between 0% and 4% according to the number of parallel trace segments), while the latency of the packets belonging to the small flows align with the best of the baseline (at low loads) or the latency of Reframer minus the buffering delay (at higher loads). Somewhat surprisingly, Reframer achieves *lower latencies* than the baseline for small flows across *all* traffic loads by simply delaying and reordering only large flows. We leave the detection of heavy-hitters, for instance detecting which flows could have generated multiple larger bursts, as future work.

5.4 Flow-Level Experiments (HTTP Server)

In this experiment, we evaluate Reframer to assess its impact on a web server application using TCP connections to dispatch files of 1MB to a set of 2048 clients continuously fetching files. By controlling the rate and number of clients' requests, we are also able to substantially increase the throughput of the test and exploit the 100G NIC interfaces. To simulate 4096 independent clients with more realistic latencies, we place a machine in-the-wire that delays packets in per-flow queues by ~10 ms±~2 ms. Hence, each connection exhibits slightly different delays, for an average ~20 ms delay. The focus of this experiment is on flow-level metrics, with the goal to check whether (*i*) Reframer improves the FCT of the dispatched files and (*ii*) the buffering delays cause any troubles to the underlying congestion control mechanism (i.e., TCP Cubic). We compare the *baseline* against Reframer. We selected NGNIX 1.14 as the web server running on 16 cores of the DUT, while Reframer runs on a dedicated NF machine using 6 cores. Reframer reorders packets in both directions, aggregates TCP ACKs from the client to the server, and eventually reorders out-of-order TCP packets. Fig. 19(a) shows that Reframer increases the application throughput by 20%. The observed improvements are due to the increase in spatial locality from 1.25 to 14. Fig. 19(b) shows that despite introducing delays in the order of microseconds, Reframer *reduces* FCT of TCP connections by fractions of a second (from 3.4 s to 3.1 s). ACK coalescing accounts for ¼ of the throughput improvements but does not affect the FCT.
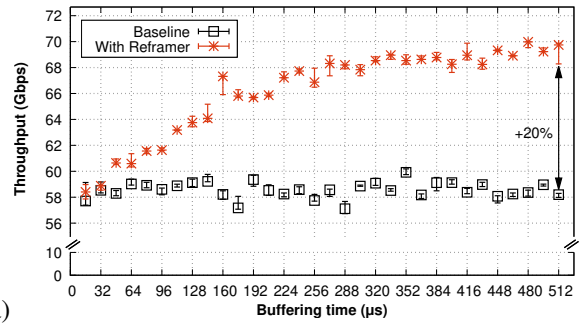
(a)
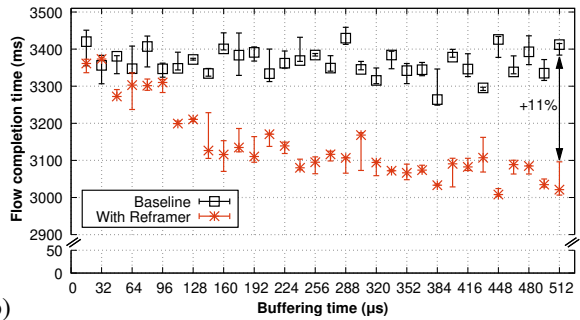
(b)

Figure 19: Impact of Reframer when reordering packets of HTTP flows: (a) Throughput and (b) FCT.

6 Related Work

Batching. Previous efforts [14, 24, 33–35] have shown the importance of processing entire batches of packets rather than individual packets (not necessarily belonging to the same flow) in order to amortize the costs of the interrupts in the NF processing system (e.g., Batchy [14], SCC [35]). Our work is orthogonal to these approaches because Reframer improves the performance of a server application in a "transparent" way, e.g., by reordering packets on the NIC or before being sent to the application. Moreover, existing packet processors do not increase the traffic locality at the per-flow level, which we show to be critical to achieve high performance in §5.2.

Traffic coalescing. Receive Side Coalescing (RSC) [36] aka LRO accelerates TCP processing by merging consecutive packets of a TCP flow into a single frame. Unfortunately, as shown in §2.2, hardware-based LRO breaks as soon as packets are interleaved. Similarly, the software implementation of LRO in the Linux kernel, called Generic Receive Offload (GRO) [37], suffers from the same problem.

Packet schedulers. We distinguish between *hardware* and *software* packet schedulers. Hardware packet schedulers typically try to realize different approximations of universal schedulers mapping packet ranks to the available queues on the hardware (e.g., [38–43]). Another set of hardware packet schedulers (e.g., [44–52]) focus on network-level optimization in datacenters (e.g., minimize traffic congestion). None of *all* these works have explicitly looked at the possibility of batching and scheduling packets to increase traffic locality at

the per-flow level. For instance, pFabric [44] would *nullify* any high traffic locality by purposely interleaving flows.

Stardust [53] is a hardware-based fabric architecture for datacenter scale networks. Stardust classifies packets per destination and chops them into bounded-size cells. This technique enables Stardust to send chops of packets in a burst, which potentially minimizes the cost of processing them at the destination. However, based on our understanding, Stardust's cells are flow-agnostic, whereas Reframer improves the performance of NFs and applications by increasing the traffic locality at the per-flow level. Moreover, Reframer *purposely* delays packets, which is the *pivotal* aspect of our proposed solution.

Software-based packet schedulers (e.g., [30, 54–61]) operate at the CPU level with the goal of dispatching the incoming flows (or coflows) of traffic to the different cores on the machine running packet processing operations. Also in this case, *none* of these approaches have explicitly looked at the impact of traffic locality on the performance of the applications receiving the packets.

To summarize, existing scheduling schemes do *not* take into consideration the impact of per-flow traffic locality. In contrast, Reframer schedules and prioritizes bursts of flows using a variety of policies while exploiting opportunities for packet coalescing and increased traffic locality.

TCP accelerations. AccelTCP [62] and Tonic [63] are dual-stack solutions that offload or generalize stateful TCP operations to NICs in order to simplify the end host stack. Such operations include connection setup and teardown as well as connection splicing that relays packets of two connections entirely within a NIC. To the best of our knowledge, *none* of these works explicitly aim to increase per-flow traffic locality.

7   Conclusions

This work unveiled the importance of packet ordering on many applications, specifically NFs and TCP applications. We showed that receiving traffic by bursts of packets of the same flow could improve a server's performance by a factor of $3\times$ as opposed to receiving packets of interleaved flows. Analyzing realistic traffic, we found that by slightly delaying traffic, even by only 64 μs, one can potentially re-build bursts of packets. We then described Reframer, a software NF, capable of re-building bursts at 28 Gbps with a single core, and scalable to 100 Gbps with a few cores. We showed Reframer is still highly beneficial when deployed as part of an NF chain, while bringing performance improvements to the server and its services. We believe this paper will spur new research around the delicate interaction between congestion control mechanisms and cache-based optimizations. It also calls for further potential improvements, e.g., decreasing the number of frames by coalescing payload or realizing Reframer in hardware.

References

[1] Intel Barefoot Networks. Tofino-2 Second-generation of World's fastest P4-programmable Ethernet switch ASICs, 2020. https://www.barefootnetworks.com/products/brief-tofino-2/.

[2] NVIDIA Mellanox. ConnectX®-6 EN IC 200GbE Ethernet Adapter IC, 2019. https://www.mellanox.com/related-docs/prod_silicon/PB_ConnectX-6_EN_IC.pdf.

[3] Zhiping Yao, Jasmeet Bagga, Hany Morsy. Introducing Backpack: Our second-generation modular open switch, November 2016. https://engineering.fb.com/data-center-engineering/introducing-backpack-our-second-generation-modular-open-switch/.

[4] Shelby Thomas, Rob McGuinness, Geoffrey M. Voelker, and George Porter. Dark packets and the end of network scaling. In *Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems*, ANCS '18, page 1–14, New York, NY, USA, 2018. Association for Computing Machinery.

[5] Shelby Thomas, Geoffrey M. Voelker, and George Porter. Cachecloud: Towards speed-of-light datacenter communication. In *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*, Boston, MA, July 2018. USENIX Association.

[6] NVIDIA Mellanox Technologies. RDMA and RoCE for Ethernet Network Efficiency Performance, 2020. https://www.mellanox.com/products/adapter-ethernet-SW/RDMA-RoCE-Ethernet-Network-Efficiency.

[7] Intel. Data Direct I/O Technology, 2017. http://www.intel.co.jp/content/dam/www/public/us/en/documents/white-papers/data-direct-i-o-technology-overview-paper.pdf.

[8] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. Reexamining Direct Cache Access to Optimize I/O Intensive Applications for Multi-hundred-gigabit Networks. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 673–689. USENIX Association, July 2020.

[9] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. Make the Most out of Last Level Cache in Intel Processors. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 8:1–8:17, New York, NY, USA, 2019. ACM. http://doi.acm.org/10.1145/3302424.3303977.

[10] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. Metron: NFV Service Chains at the True Speed of the Underlying Hardware. In *15th USENIX Conference on Networked Systems Design and Implementation*, NSDI'18, pages 171–186, Renton, WA, 2018. USENIX Association. https://www.usenix.org/system/files/conference/nsdi18/nsdi18-katsikas.pdf.

[11] Alireza Farshin, Tom Barbette, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. PacketMill: Toward per-core 100-Gbps Networking. In *Proceedings of the Twenty-Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, New York, NY, USA, 2021. Association for Computing Machinery.

[12] Yifan Yuan, Yipeng Wang, Ren Wang, and Jian Huang. HALO: Accelerating Flow Classification for Scalable Packet Processing in NFV. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pages 601–614, New York, NY, USA, 2019. ACM. http://doi.acm.org/10.1145/3307650.3322272.

[13] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. Bbr: Congestion-based congestion control. *ACM Queue*, 14, September-October:20 – 53, 2016.

[14] Tamás Lévai, Felicián Németh, Barath Raghavan, and Gabor Retvari. Batchy: Batch-scheduling Data Flow Graphs with Service-level Objectives. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 633–649, Santa Clara, CA, February 2020. USENIX Association. https://www.usenix.org/conference/nsdi20/presentation/levai.

[15] iPerf - The ultimate speed test tool for TCP, UDP and SCTP. https://iperf.fr/.

[16] Open vSwitch. An Open Virtual Switch. http://openvswitch.org.

[17] VMware. NSX-T Data Center Documentation, 2020. https://docs.vmware.com/en/VMware-NSX-T-Data-Center/index.html.

[18] OpenStack. Open Source Cloud Computing Software, 2020. https://www.openstack.org/.

[19] Red Hat. OpenShift - The Kubernetes platform for big ideas, 2020. https://www.openshift.com/.

[20] The Linux Foundation. Kubernetes, 2020. https://kubernetes.io/.

[21] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J. Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Jonathan Stringer, Pravin Shelar, Keith Amidon, and Martín Casado. The Design and Implementation of Open vSwitch. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 117–130, Berkeley, CA, USA, 2015. USENIX Association. https://www.usenix.org/system/files/conference/nsdi15/nsdi15-paper-pfaff.pdf.

[22] V. Srinivasan, S. Suri, and G. Varghese. Packet Classification Using Tuple Space Search. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '99, page 135–146, New York, NY, USA, 1999. Association for Computing Machinery. https://doi.org/10.1145/316188.316216.

[23] Andrew Theurer, Red Hat. Testing the Performance Impact of the Exact Match Cache. In *Open vSwitch 2018 Fall Conference*, pages 1–17, San Jose, CA, USA, December 2018. https://www.openvswitch.org/support/ovscon2018/5/1330-theurer.pdf.

[24] Tom Barbette, Cyril Soldani, and Laurent Mathy. Fast userspace packet processing. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '15, pages 5–16, Washington, DC, USA, May 2015. IEEE Computer Society. http://dl.acm.org/citation.cfm?id=2772722.2772727.

[25] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: congestion-based congestion control. *ACM Queue*, 14(5):20–53, 2016. http://doi.acm.org/10.1145/3012426.3022184.

[26] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. *SIGCOMM Comput. Commun. Rev.*,

45(4):537–550, August 2015. https://doi.org/10.1145/2829988.2787510.

[27] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 19–19, Berkeley, CA, USA, 2012. USENIX Association. http://dl.acm.org/citation.cfm?id=2228298.2228324.

[28] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Vinh The Lam, Carlo Contavalli, and Amin Vahdat. Carousel: Scalable traffic shaping at end hosts. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 404–417, New York, NY, USA, 2017. ACM. http://doi.acm.org/10.1145/3098822.3098852.

[29] Allen B. Downey. TCP Self-Clocking and Bandwidth Sharing. *Comput. Netw.*, 51(13):3844–3863, September 2007. https://doi.org/10.1016/j.comnet.2007.04.005.

[30] Tom Barbette, Cyril Soldani, Romain Gaillard, and Laurent Mathy. Building a chain of high-speed VNFs in no time. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, Bucharest, Romania, June 2018. IEEE. https://doi.org/10.1109/HPSR.2018.8850742.

[31] Will Glozer. wrk - a HTTP benchmarking tool. https://github.com/wg/wrk.

[32] Linux Foundation. Data Plane Development Kit (DPDK), 2020. http://www.dpdk.org.

[33] Luigi Rizzo. netmap: A novel framework for fast packet i/o. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2012. http://info.iet.unipi.it/~luigi/netmap/.

[34] Joongi Kim, Seonggu Huh, Keon Jang, KyoungSoo Park, and Sue Moon. The power of batching in the Click modular router. In *Proceedings of the ACM Asia-Pacific Workshop on Systems (APSYS)*, 2012. http://doi.acm.org/10.1145/2349896.2349910.

[35] Georgios P. Katsikas, Gerald Q. Maguire Jr., and Dejan Kostić. Profiling and accelerating commodity NFV service chains with SCC. *Journal of Systems and Software*, 127C:12–27, February 2017. https://doi.org/10.1016/j.jss.2017.01.005.

[36] Srihari Makineni, Ravi Iyer, Partha Sarangam, Donald Newell, Li Zhao, Ramesh Illikkal, and Jaideep Moses. Receive Side Coalescing for Accelerating TCP/IP Processing. In *Proceedings of the 13th International Conference on High Performance Computing*, HiPC'06, pages 289–300, Berlin, Heidelberg, 2006. Springer-Verlag. http://dx.doi.org/10.1007/11945918_31.

[37] Jonathan Corbet. Generic receive offload, 2009. https://lwn.net/Articles/358910/.

[38] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. Programmable Packet Scheduling at Line Rate. In *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference*, SIGCOMM '16, pages 44–57, New York, NY, USA, 2016. ACM. http://doi.acm.org/10.1145/2934872.2934899.

[39] Radhika Mittal, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Universal Packet Scheduling. In *Proceedings of the 13th USENIX Conference on Networked Systems Design and Implementation*, NSDI'16, pages 501–521, Berkeley, CA, USA, 2016. USENIX Association. http://dl.acm.org/citation.cfm?id=2930611.2930644.

[40] Vishal Shrivastav. Fast, scalable, and programmable packet scheduler in hardware. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 367–379, New York, NY, USA, 2019. Association for Computing Machinery. https://doi.org/10.1145/3341302.3342090.

[41] Albert Gran Alcoz, Alexander Dietmüller, and Laurent Vanbever. SP-PIFO: Approximating Push-In First-Out Behaviors using Strict-Priority Queues. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 59–76, Santa Clara, CA, February 2020. USENIX Association. https://www.usenix.org/conference/nsdi20/presentation/alcoz.

[42] Naveen Kr. Sharma, Chenxingyu Zhao, Ming Liu, Pravein G Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. Programmable Calendar Queues for High-speed Packet Scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 685–699, Santa Clara, CA, February 2020. USENIX Association. https://www.usenix.org/conference/nsdi20/presentation/sharma.

[43] Brent Stephens, Aditya Akella, and Michael Swift. Loom: Flexible and Efficient NIC Packet Scheduling. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages

33–46, Boston, MA, February 2019. USENIX Association. https://www.usenix.org/conference/nsdi19/presentation/stephens.

[44] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. PFabric: Minimal near-Optimal Datacenter Transport. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 435–446, New York, NY, USA, 2013. Association for Computing Machinery. https://doi.org/10.1145/2486001.2486031.

[45] Y. Lu, G. Chen, L. Luo, K. Tan, Y. Xiong, X. Wang, and E. Chen. One more queue is enough: Minimizing flow completion time with explicit priority notification. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pages 1–9, 2017. https://doi.org/10.1109/INFOCOM.2017.8056946.

[46] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better Never than Late: Meeting Deadlines in Datacenter Networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, page 50–61, New York, NY, USA, 2011. Association for Computing Machinery. https://doi.org/10.1145/2018436.2018443.

[47] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A Centralized "Zero-Queue" Datacenter Network. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 307–318, New York, NY, USA, 2014. Association for Computing Machinery. https://doi.org/10.1145/2619239.2626309.

[48] Peter X. Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. PHost: Distributed near-Optimal Datacenter Transport over Commodity Network Fabric. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15, New York, NY, USA, 2015. Association for Computing Machinery. https://doi.org/10.1145/2716281.2836086.

[49] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 221–235, New York, NY, USA, 2018. Association for Computing Machinery. https://doi.org/10.1145/3230543.3230564.

[50] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient Coflow Scheduling with Varys. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIG-COMM '14, page 443–454, New York, NY, USA, 2014. Association for Computing Machinery. https://doi.org/10.1145/2619239.2626315.

[51] Saksham Agarwal, Shijin Rajakrishnan, Akshay Narayan, Rachit Agarwal, David Shmoys, and Amin Vahdat. Sincronia: Near-Optimal Network Design for Coflows. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 16–29, New York, NY, USA, 2018. Association for Computing Machinery. https://doi.org/10.1145/3230543.3230569.

[52] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. Information-agnostic flow scheduling for commodity data centers. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, page 455–468, USA, 2015. USENIX Association.

[53] Noa Zilberman, Gabi Bracha, and Golan Schzukin. Stardust: Divide and conquer in the data center network. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 141–160, 2019.

[54] Ahmed Saeed, Yimeng Zhao, Nandita Dukkipati, Ellen Zegura, Mostafa Ammar, Khaled Harras, and Amin Vahdat. Eiffel: Efficient and Flexible Software Packet Scheduling. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 17–32, Boston, MA, February 2019. USENIX Association. https://www.usenix.org/conference/nsdi19/presentation/saeed.

[55] Vojislav Đukić, Sangeetha Abdu Jyothi, Bojan Karlas, Muhsen Owaida, Ce Zhang, and Ankit Singla. Is advance knowledge of flow sizes a plausible assumption? In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 565–580, Boston, MA, February 2019. USENIX Association. https://www.usenix.org/conference/nsdi19/presentation/dukic.

[56] Sameer G. Kulkarni, Wei Zhang, Jinho Hwang, Shriram Rajagopalan, K. K. Ramakrishnan, Timothy Wood, Mayutan Arumaithurai, and Xiaoming Fu. NFVnice: Dynamic Backpressure and Scheduling for NFV Service Chains. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 71–84, New York, NY, USA, 2017. ACM. http://doi.acm.org/10.1145/3098822.3098828.

[57] Guikai Liu and Qing Li. Fair and Efficient Packet Scheduling Using Resilient Quantum Round-Robin. *Journal of Networks*, 9(2):269–276, 2014. https://doi.org/10.4304/jnw.9.2.269-276.

[58] Anat Bremler-Barr, Yotam Harchol, and David Hay. OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions. In *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference*, SIGCOMM '16, pages 511–524, New York, NY, USA, 2016. ACM. http://doi.acm.org/10.1145/2934872.2934875.

[59] Georgios P. Katsikas, Marcel Enguehard, Maciej Kuź-niar, Gerald Q. Maguire Jr., and Dejan Kostić. SNF: Synthesizing high performance NFV service chains. *PeerJ Computer Science*, 2:e98, November 2016. http://dx.doi.org/10.7717/peerj-cs.98.

[60] Tom Barbette, Georgios P. Katsikas, Gerald Q. Maguire, Jr., and Dejan Kostić. RSS++: load and state-aware receive side scaling. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, CoNEXT '19, pages 318–333, New York, NY, USA, 2019. ACM. http://doi.acm.org/10.1145/3359989.3365412.

[61] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Gerald Q. Maguire Jr., and Rebecca Steinert. Metron: High-Performance NFV Service Chaining Even in the Presence of Blackboxes. *ACM Trans. Comput. Syst.*, 38(1–2), July 2021. https://doi.org/10.1145/3465628.

[62] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. AccelTCP: Accelerating Network Applications with Stateful TCP Offloading. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 77–92, Santa Clara, CA, February 2020. USENIX Association. https://www.usenix.org/conference/nsdi20/presentation/moon.

[63] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling programmable transport protocols in high-speed nics. In Ranjita Bhagwan and George Porter, editors, *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 93–109. USENIX Association, 2020.
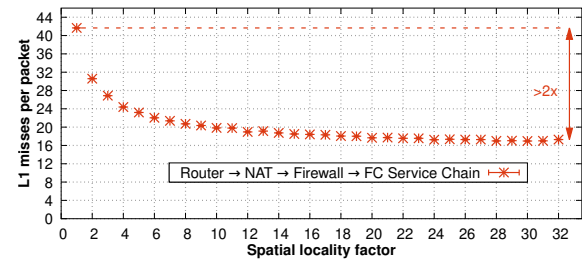
A Supplementary Material

This section provides some additional material for this paper.
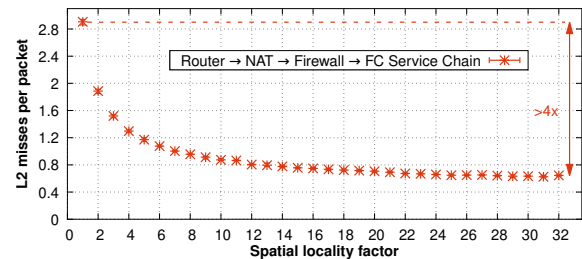
A.1 Deploying a chain of NFs

In addition to experiments discussed in §2.4, we deployed a chain of network functions on the DUT as a complementary experiment. In this test, we connected a Router, a NAT, a
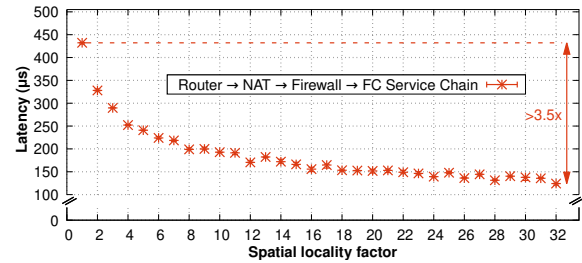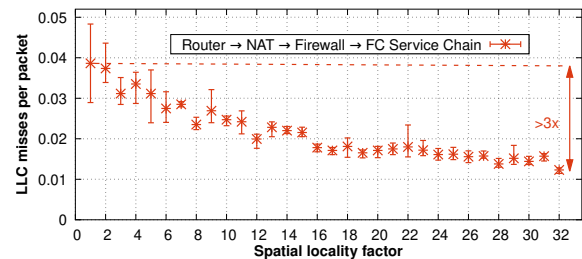

(a) End-to-end latency in µs.

(b) L1 misses per packet.

(c) L2 misses per packet.

(d) LLC misses per packet.

Figure 20: Impact of packet order on the performance of a Router→NAT→Firewall→FC chain of NFs.

firewall, and a Flow statistics counter (FC) in a row, as a chain of NFs. The DUT uses 4 CPU cores to serve the packets and it is implemented in a run-to-completion model to exploit the parallelism on the processors. All the other configurations are similar to §2.4.

Since the deployed chain is both CPU and memory intensive, the scale of CPU cycles per packet and the end-to-end latency are higher in compare to individual NAT and firewall experiments in §2.4. However, the results in Figure 20 confirm that, regardless of the complexity of the implemented
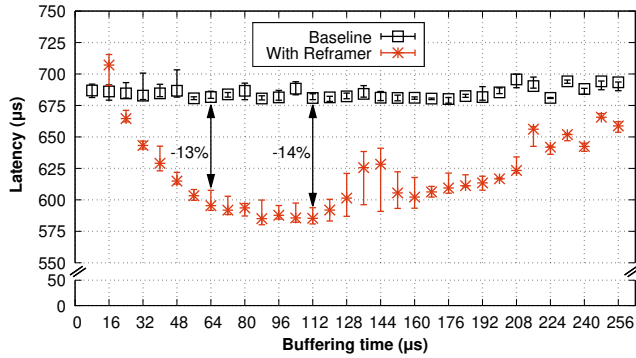
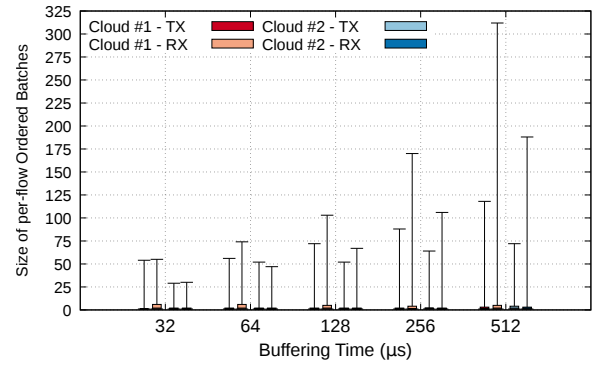Figure 21: Latency of Reframer when offloaded into a Smart NIC which precedes the NF chain.
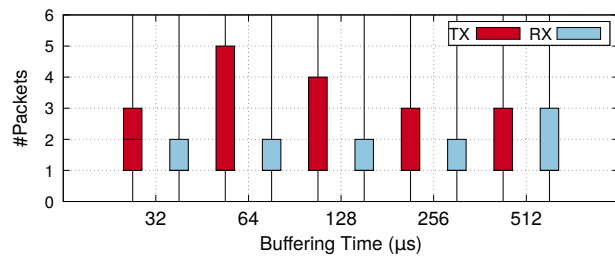

Figure 22: Impact of increasing the waiting time on the probability of coalescing Transmission Control Protocol (TCP) ACKs. (The figure is intentionally scaled to enhance the visibility of the first to third quartiles.)

NFs, ordering the packets has a significant impact on the DUT's performance.

Similar to §2.4, the fundamental reason of such enhancement is efficient utilization of system caches. In this experiment, since the DUT needs more data to process a packet, the improvement in cache misses has been extended to L2 and LLC. Figure 20d shows a substantial improvement in terms of number of LLC misses. Note that this improvement is not happening only in LLC. We also can see the same trend (with smaller improvement factors) in L1 (Figure 20b) and L2 (Figure 20c) caches.

A.2 Running Reframer in a SmartNIC

Figure 21 shows the latency induced by the Reframer versus a baseline NF, when deployed on two Arm cores of a Mellanox Bluefield SmartNIC.

A.3 Analyzing the Trace

To perform the analysis, we have used PcapPlusPlus to create a CSV file composed of useful fields. Then, we split the 62-GB file to per-flow CSV files via Spark. Finally, we use Python data science libraries (e.g., Pandas and NumPy) to calculate the probability of receiving different batch sizes

Figure 23: Impact of increasing the waiting time on the probability of receiving packets with the same TCP flow.

within different time windows. Listing 1 shows the python code used to process each flow.

ACK coalescing. Fig. 22 shows the potential improvement from TCP ACK coalescing in the campus trace. We calculated the distribution of the number of per-flow packets with enabled TCP acknowledgment flag (ACK) within a time frame.

```python
import pandas as pd
import numpy as np

# Calculate the size of ordered batches for each flow
# based on the input window size (ws)
def process(flow,ws):
    # Getting the timestamp of packets in a flow
    ts = flow['ts'].to_numpy()

    # Initialize variables
    batch_size=1
    i=1
    ordered_size=np.empty((0))
    threshold=32

    # Check the size of flow
    if ts.size == 1:
        # Add the batch_size to the array of ordered_size
        ordered_size = np.append(ordered_size,batch_size)
    else:
        # Sort the timestamps
        sorted_ts = np.sort(ts)
        # Start from the first packet of a flow
        base_ts = sorted_ts[0]

        # Continue while there is still more packets
        while i < sorted_ts.size:

            # Increase the size as long as the next packet
            # arrives before the end of the window size
            if sorted_ts[i] - base_ts < ws :
                batch_size = batch_size + 1

            # If the size of the batch is larger than
            # the threshold or the next packet of the flow
            # comes after the end of the window size.
            # The batch size and the time counter,
            # we should stop the time counter. Stopping
            # the counter means that we start the counter
            # again with the next packet. Also, we update the
            # beginning of the window size with the timestamp
            # of the newly arrived packet.
            if (batch_size >=threshold) \
                or (sorted_ts[i] - base_ts >= ws):
                # Update the beginning of the window size
                base_ts = sorted_ts[i]
                # Add the batch size to the array
                ordered_size = np.append(ordered_size,batch_size)

                # Reset the batch size
                if batch_size != 1:
                    batch_size = 1
                    i = i + 1
                    ordered_size = np.append(ordered_size,count)
    return ordered_size
```

Listing 1: Python function used to calculate the size of the per-flow batches.

Table 1: Flow statistics per Internet Protocol (IP) address of two popular cloud providers.

IP		#Flows		Flow Size (#Packets)							
				Min		Mean		Median		Max	
		TX	RX	TX	RX	TX	RX	TX	RX	TX	RX
Cloud-1	IP-1	19985	20039	1	1	47.45	54.08	16	16	87131	221594
	IP-2	5384	5433	1	1	14.92	19.11	13	15	92	157
	IP-3	4741	4748	1	1	42.55	51.58	15	15	43500	32540
	IP-4	4567	4564	1	1	52.62	37.18	16	16	38515	19168
	IP-5	4245	3805	1	1	187.12	1397.09	8	29	57392	217309
	IP-6	4155	4000	1	1	12.83	13.63	12	10	831	2775
	IP-7	3980	3958	1	1	13.48	9.63	11	8	663	394
	IP-8	3759	3759	2	2	258.30	318.38	11	10	33403	33356
	IP-9	3154	3159	1	1	28.86	21.89	14	10	310	279
	IP-10	2996	2984	1	1	39.76	22.48	56	28	238	228
Cloud-2	IP-1	19776	19762	1	1	165.77	137.07	10	10	1615124	764636
	IP-2	18120	18103	1	1	19.26	44.42	10	10	17929	42637
	IP-3	15967	15945	1	1	39.33	68.71	11	11	68306	92210
	IP-4	11168	11150	1	1	105.09	62.86	10	10	255327	121520
	IP-5	9207	9235	1	1	110.75	119.08	21	20	9129	7665
	IP-6	8828	8803	1	1	521.57	265.83	10	10	1353933	587496
	IP-7	5897	5879	1	1	51.44	67.54	15	14	12448	13422
	IP-8	5330	4993	1	1	42.93	77.05	12	11	7248	18137
	IP-9	4499	4479	1	1	116.08	198.77	16	15	16625	37459
	IP-10	3785	3775	1	1	57.22	75.43	17	16	4371	8287
	IP-11	3369	3362	1	1	235.40	306.51	17	15	19614	22311
	IP-12	3355	3279	1	1	1501.01	493.11	18	19	4905771	1409104
	IP-13	3152	3144	1	1	23.20	33.87	11	10	867	2617
	IP-14	3113	3078	1	1	28.69	47.82	15	14	2952	7922
	IP-15	2864	2868	1	1	59.26	83.18	12	12	9143	22045

Buffer-based End-to-end Request Event Monitoring in the Cloud

Kaihui Gao[*][†], Chen Sun[†], Shuai Wang[*], Dan Li[*],
Yu Zhou[†], Hongqiang Harry Liu[†], Lingjun Zhu[†], Ming Zhang[†]
[*]*Tsinghua University*   [†]*Alibaba Group*

Abstract

Request latency is a crucial concern for modern cloud providers. Due to various causes in hosts and networks, requests can suffer from request latency anomalies (RLAs), which may violate the Service-Level Agreement for tenants. However, existing performance monitoring tools have incomplete coverage and inconsistent semantics for monitoring requests, resulting in the difficulty to accurately diagnose RLAs.

This paper presents BufScope, a high-coverage request event monitoring system, which aims to capture most RLA-related events with consistent request-level semantics in the end-to-end datapath of request. BufScope models the datapath of request as a buffer chain and defines RLA-related events based on different properties of buffers, so as to *end-to-end monitor* the root causes of RLA. To achieve *consistent semantics* for captured events, BufScope designs a concise request-level semantic injection mechanism to make events captured in networks have the victim requests' ID, and offloads the realization to SmartNICs for *low overhead*. We have implemented BufScope on commodity SmartNICs and programmable switches. Evaluation results show that BufScope can diagnose 95% RLAs with <0.07% network bandwidth overhead and <1% application throughput decline.

1 Introduction

With the emergence of cloud-native architecture [1], application-layer requests (*e.g.,* RPC, HTTP, and RESTful requests) become a fundamental component in the cloud [2]. The request latency is the total elapsed time across a request end-to-end datapath, including the application, the end-host network stack and the underlying network. Since request latency directly affects the performance of many distributed applications [3], it has become a crucial concern [4–6] for cloud providers. Besides, request-level information is the tie between the tenants and the cloud providers. For instance, when a request (*e.g.,* search, storage I/O) encounters a surge of latency, the tenant will provide the operators with the request-level descriptive information to diagnose the anomaly [7].

Request latency anomalies (RLAs), which cause long-tailed request latency, are not rare in clouds. According to the data of a block storage cluster with over 40,000 servers from a prominent global cloud provider *Alibaba*, we observe that across all the 440 million RPC requests in one hour, 0.01% of them (44K) suffer from a latency of >100 ms, which violates the Service-Level Agreement (SLA) of the storage service. Cloud providers need to accurately diagnose RLAs to explain SLA violations, otherwise revenue loss will be caused [8].

However, accurately diagnosing RLAs is challenging, because it requires high coverage for request-level abnormal events (*i.e., request events*). Specifically, cloud providers must be able to capture as many abnormal events that happen on the *end-to-end* datapath of requests as possible, *e.g.,* data pause, congestion, drop, *etc.*, which are direct triggers of RLAs. Moreover, the captured events should be mapped to *request-level semantics* (*e.g.,* RPC ID), rather than the flow- or packet-level. In practice, it is non-trivial to extract the request-level semantics from flow- or packet-level data.

Unfortunately, existing performance or latency monitoring tools are far from satisfying the preceding requirements for diagnosing RLAs. Specifically, though distributed application performance tracing tools [9–16] can provide any request-level timing data, they cannot capture the request events below the application layer; Network performance monitoring tools [8, 17–25] can capture flow-level events that happen in the network stack or the underlying networks, such as delayed ACK, packet drop, and path change, but the captured events have no request-level information, so the events captured in applications and networks cannot be associated.

Since these existing monitoring tools have incomplete coverage and inconsistent semantics, the cloud providers often get into trouble when diagnosing RLAs. For instance, based on the RLA information reported by the tracing tool, the application owners or tenants often naturally blame server and network team [8, 24]. However, due to the mismatch of monitoring semantics, these teams have to associate the events obtained from their own monitoring tools with the RLA through coarse-grained time-correlation methods [24], which is not

only inefficient, but also inaccurate (§2.1).

The fundamental reason why existing tools fail to achieve the high coverage is that the traditional data plane in datacenter networks is a black box [8]. The request events which happen in the data plane cannot be detected and parsed as flexibly as those in the host software. Consequently, network monitoring tools typically capture accessible and coarse-grained flow- or packet-level events. This makes it difficult to end-to-end monitor RLAs with consistent request-level semantics. Fortunately, recent advances on the commodity programmable data plane provides a new foundation to improve the situation.

This paper presents BufScope, a high-coverage request event monitoring system. The main idea of BufScope is to translate most RLAs to buffer-related abnormal events, monitor all buffers in the request's end-to-end datapath, and capture all buffer-related abnormal events with consistent request-level semantics. Specifically, BufScope achieves *end-to-end monitoring* and *consistent request-level semantics* through two core designs which keep *low overhead* in mind.

(i) Buffer event modeling. The main purpose of buffer is to deal with the mismatch between upstream and downstream processing rates. If upstream or downstream processing has an anomaly, the buffer will reveal the corresponding abnormal events, such as queue buildup, data pause, and packet out-of-order [26]. Based on the operational experience in *Alibaba*, we observe that most (>90%) RLAs reveal abnormal events in buffers (§3.1). The remaining (<10%) RLAs that come from NIC flapping, link corruption, bugs, *etc.*, are very difficult and inefficient to cover. For low overhead consideration, in this work we only cover RLAs with buffer-related abnormal events. Thus, BufScope models the end-to-end datapath of request as a buffer chain (§3.2), including the application, network stack, NIC and switch, and monitors RLA-related abnormal events that happen in all the buffers.

However, these buffers may have different RLA-related abnormal events, and pre-defining all the events for all types of buffers is challenging. For example, in lossy Ethernet, packets may be dropped before entering the buffer, while in lossless Ethernet, the upstream switch will pause packet forwarding to avoid the packet drop in the downstream switch. In response, BufScope uniformly classifies all buffers in both hosts and networks according to three properties, including priority awareness, order sensitivity, and enqueue feature. Based on these properties, BufScope defines a complete buffer event library, including packet drops, congestion, pause, out-of-order and priority contention (§3.3). Then, operators can monitor the corresponding events in a buffer based on its properties.

(ii) Request-level semantic injection. The lack of request-level semantics in the network is because the request header may not exist in the packet payload; even if it exists, its location in the payload is not fixed. This causes commodity programmable switch to fail to extract the request-level information when generating abnormal events. In response, BufScope designs a concise semantic injection mechanism,

which just inserts the offset of the first request header (if it is in the payload) at the end of the packet header (§3.4). Then, based on the location-specific information, programmable switches can iteratively parse all request identifier in a packet.

A straightforward approach to injecting request-level semantic is to implement the function in the end-host network stack. However, the overhead of this strawman design is significant for applications that adopt run-to-completion (RTC) model (§5.3). RTC model packs the entire logic (including application and network stack processing) in one single thread to achieve ultra-low latency, which is quite common for large-scale datacenter applications [7, 27]. To reduce the impact of request-level semantic injection on the application performance, we offload the operation to hardware.

We have integrated BufScope in an open-source RPC system and *Alibaba*'s production storage application with Broadcom PS225 SmartNICs and Barefoot Tofino switches. Testbed-based evaluation shows that BufScope can diagnose 95% RLAs (64% for the combined solution of existing state-of-the-art monitoring tools [8, 15, 21]) with <0.07% network bandwidth overhead (>4% for the baseline) and <1% application throughput decline (4.3% for the baseline).

2 Background and Motivation

In this section, we firstly use representative experiences of *Alibaba* to demonstrate the RLAs in production. Then, we analyze the limitations of existing monitoring tools to accurately diagnose RLAs. Finally, we present this paper's motivation.

2.1 RLAs in the Cloud

There are generally two types of performance anomalies for one request: connectivity loss and RLA. The former means the client loses connectivity to the remote server for seconds to minutes, due to issues such as hardware failure, program corruption, or network outage. The latter type of anomaly means that, even though the request can be finally completed, the latency for this request is larger than expected, which compromises the SLA. We focus on the monitoring of RLA, which is easy to happen but very difficult to mitigate. This is because RLAs are usually caused by abnormal events in a shorter time-scale with randomness, leaving minor fingerprints for monitoring and locating. Potential root causes could be polling hang and badly-tuned network stack parameters in hosts, congestion and packet drop in networks [8, 17, 24], *etc.*

To understand the impact of RLA on application performance, we have conducted experiments using the *Alibaba*'s production block storage application, which adopts RPC framework, user-level network stack and RTC model. One front-end server constantly performs 4KB file read operations from the storage back-end over RPC request. We simulate RLAs by adding microsecond-level latency to the RPC pro-
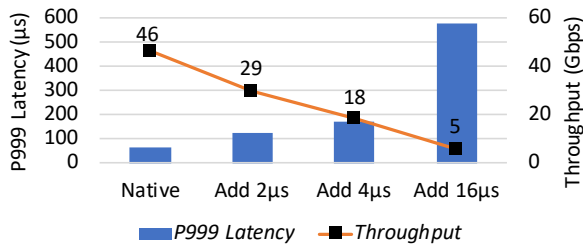
Figure 1: Impact of RLAs on application performance.

cessing logic, and measure its affection on the application regarding to end-to-end tail latency and overall throughput.

As shown in Figure 1, a 2μs latency added to every single RPC could be amplified to around 50μs increased end-to-end tail latency, and jeopardize the overall throughput by up to 36.9%, which are even worse under severe RLA. The reason why RLAs could severely compromise the application performance is that once the logic processes RPCs slower than NIC bandwidth (50Gbps in this experiment), lots of packets would jam the NIC buffers and be dropped, causing the network stack to retransmit massive packets and slow down, and leading to severe performance decline.

Such a performance decline would violate the SLA of cloud service, which requires an explanation in practice. However, cloud providers often face difficulties in the explanation, we list two representative real cases to show this.

Case-1: Is the network congestion causing the RLA? A tenant reported persistently low transmission rate between two VMs. The tenant naturally blames the network, since network congestion could slow down the sending rate. Then, network operators first retrieved switch queue and drop statistics. Data showed that the network utilization remained low and no packets were lost during that period. They have to reproduce the case using the VM's trace. However, the real cause is the limited TCP receive buffer in the host, which may be caused by CPU polling hang. It is hard for the network operators to claim their innocence unless they could detect the real cause by end-to-end monitoring.

Case-2: Is the cause in the network or end-host? A tenant reported an unexpected latency glitches (100s of ms) of a read request. To diagnose the RLA, the storage application owners first checked the request's trace obtained by their tracing tool and found that the interval between the request send and receive exceeded the expectation. Then, the application owners passed the ticket to the server and network team. Operators of the server team look up the *second-level* logs of kernel, CPU *etc.*, based on the timestamp in the ticket; Operators of the network team query the *flow-level* monitoring system if packet losses or network faults have occurred in the request's flow. Even if these queries have results, neither team has high confidence to claim their innocence due to the mismatch of monitoring granularity among these teams.

The above two cases indicate that cloud providers need a confident and accurate end-to-end monitoring tool to improve the efficiency of the explanation for SLA violations.

2.2 Limitations of Existing Monitoring Tools

From the two cases above, we can also see that the challenges of RLA diagnosis stem from the variety of the locations of root causes. Thus, accurate RLA diagnosis requires monitoring tools to have high coverage for the causes. However, existing monitoring tools have two limitations to achieve it:

(i) Incomplete coverage. Existing tools monitor partial datapath of requests, which either focus on application layer tracing/logging [3, 9–11, 16, 28–30], transport layer monitoring [17, 21, 24, 25, 31], network monitoring [8, 18, 19, 26, 32–35], or partial combination [34, 36–38]. In addition, existing tools define a separate set of abnormal events based on their monitoring goals. For example, Dapper [21] infers TCP performance events (*e.g.,* non-backlogged, congestion and delayed ACK) by analyzing packet statistics; Trumpet [23] leverages triggers at end-hosts to monitor network-wide events (*e.g.,* burst, heavy flow and congestion); NetSeer [8] monitors flow-level abnormal events (*e.g.,* packet drop, queuing, detouring and pause) in networks; performance profiling tools (*e.g.,* Perf [39]) can analyze events (*e.g.,* CPU cycles, page fault and cache miss) that occur during program execution; tracing tools [9–11, 14, 15] record timing data about requests and provide API to monitor application-specific annotations. Overall, there is no tool that can capture all RLA-related events in the end-to-end datapath of request so far, causing that operators have no confidence to claim their innocence (*e.g.,* Case-1).

(ii) Inconsistent semantics. Since these existing tools have different focuses, cloud providers have to combine multiple monitoring tools in production to cover the datapath of request as fully as possible. For example, tracing [9–11, 14, 15] is used in the application layer to track the performance of requests, and network monitoring tools [8, 18, 19] are used in the underlying network to record flow-level abnormal events. However, these monitoring tools have inconsistent semantics, the abnormal events captured by them cannot be correlated, leading to the failure of this combination (*e.g.,* Case-2). We obtain the production storage application and anonymized request traces, and run them on our testbed for 6 hours (§5.1). We use existing monitoring tools to capture abnormal events during that period and try to diagnose the cause of detected slow RPCs. Unfortunately, existing monitoring tools fail to explain a large portion of slow RPCs. First, request-level events and timing data collected by application tracing tool are too coarse-grained and incomplete, and can only explain 28% RLAs. Then, based on the time-correlation methods, flow-level events captured by network monitors can only infer 36% more RLAs, leaving 36% RLAs inexplicable.

2.3 Motivation

To accurately diagnose all RLAs, it is necessary to monitor the entire life cycle of all individual requests. Thus, our goal is to design a high-coverage request event monitoring sys-

tem which can monitor RLA-related events with consistent request-level semantic in the end-to-end datapath of request.

The fundamental limitation of existing performance monitoring tools to achieve our goal is that, they cannot uniformly model the data plane in network hardware and the datapath in end-host software. Unlike the software, traditional fixed-function data plane in network only provides limited accessibility for packets and black-box visibility [8].

With the development of commodity programmable hardware, which has been widely deployed in modern cloud, we see the opportunity of completely realizing our goal. We believe this choice is rational because of two unique advantages of programmable hardware. First, with the help of programmable switches and NICs, fine-grained abnormal events in networks can be easily detected, parsed and reported [8]. It makes monitoring the data plane in networks as flexible as monitoring the datapath in software. Second, SmartNICs show promising capability to offload CPU-consumption tasks [40, 41]. By leveraging them, we can achieve the consistent request-level monitoring semantics with low overhead.

3 Design

This section first outlines the overview of BufScope, then elaborates BufScope's design to achieve high coverage with low overhead. Finally, it shows how cloud providers can use events captured by BufScope to diagnose and mitigate RLAs.

3.1 Overview

The crux to make operators accurately and confidently judge where and how a request gets disturbed is to track the RLA-related events that directly happen to the request's traffic.

Insight. To understand the distribution of the RLA's causes and corresponding abnormal events, we have analyze almost 500 typical incident tickets of one-day's RLAs from *Alibaba*'s block storage service, which were troubleshooted by manual debugging. We present the root causes and the locations exposed anomaly in Figure 2. The root causes spread across the datapath of request, such as polling hang in hosts, incast in NICs, and burst in switches. We derive our insight that most (>90%) RLAs expose anomalies in buffers. The remaining (<10%) RLAs come from NIC flapping, network update, bugs, *etc.*, which requires hardware- or program-specific monitoring, and is hard to cover using a general solution. Besides, given that buffers are where the request's traffic stays and latency rises [26], BufScope chooses the buffer as the key object to monitor the most RLA-related events.

Design goals. BufScope is a request event monitoring system, which aims to achieve high coverage for RLAs' root causes. Specifically, the design of BufScope needs to achieve the following three requirements. First, BufScope should be able to monitor the request's end-to-end datapath for RLA-related events. Second, all events captured by BufScope need to have
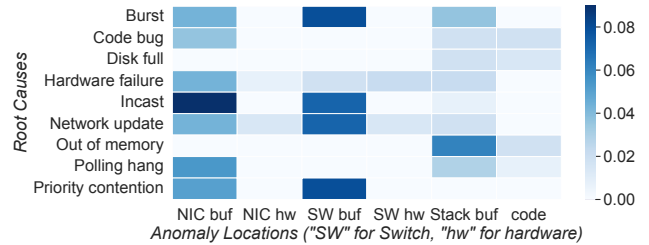

Figure 2: Heatmap for root causes and anomaly locations of RLAs.

consistent request-level semantics to correlate the events happening in the hosts and networks. Finally, in order to reduce the impact on the performance of the monitored application, BufScope must be designed with low overhead.

Challenges. It is highly challenging to achieve the above requirements:

- *End-to-end monitoring:* For generality, BufScope needs to model a unified buffer chain for various communication frameworks and underlying networks. However, buffers are various and have different RLA-related events. Therefore, BufScope needs to define a complete event library, which contains all events that will occur in all types of buffers.

- *Consistent request-level semantics:* The uncertainty of location of request header in packet makes it hard for the commodity programmable switches to parse out the request-level information when generating events. Thus, BufScope needs to design a novel mechanism to inject request-level semantics into the specific location of packets.

- *Low overhead:* The semantic injecting mechanism of the strawman solution consumes valuable CPU resources for the RTC application, and degrade the application performance [10]. Thus, BufScope must be designed to reduce the semantic injecting overhead as much as possible.

Architecture. We present BufScope's architecture in Figure 3. Following the buffer chain model (§3.2), BufScope's agents monitor buffers along the datapath of request, including applications, network stack, NICs and switches, and capture the corresponding events according to the type of buffers (§3.3). Besides, in order to record the victim request identifier when generating events in networks, BufScope enables request-level semantic injection in sender side, and offloads it into SmartNICs to reduce overhead (§3.4). For efficient event collection (§3.3), in the software and SmartNIC, the BufScope agents execute event collection asynchronously with respect to the monitored application; In the programmable switch, after events are generated by the detection logics in the ingress and egress pipeline, the egress agent hands the events to the switch control plane for further processing, such as deduplication, batching and reporting. Finally, events from these components are associated in *Event Collector* based on the request identifier, to diagnose RLAs (§3.5).

Figure 3: The architecture of BufScope.

3.2 Buffer Chain Modeling

Buffer in both of hosts and networks is where the request's traffic stays and is the main source of abnormal request latency rising [26], which is also proved by the production data of *Alibaba*. Thus, our key design choice is ignoring the complexity of programs, function calls and hardware faults, and closely monitor all buffers in the end-to-end datapath of requests instead, to cover the most RLA-related events.

The first step is to identify the buffers in the datapath of highly diversified Layer-7 frameworks [2, 6, 42, 43]. Existing frameworks rely on different network stacks (such as kernel-bypass network stack [44], and kernel-based network stack [43]) and different threading models (such as run-to-completion model and pipeline model). To maintain its generality, BufScope is challenged to model the datapath of various Layer-7 frameworks as a uniform composition of buffers.

We address the challenge by analyzing programs of various available Layer-7 frameworks, buffers, and their connections across the end-to-end datapath of request. We observe that one single request follows one unified chain of buffers across its entire life cycle. Therefore, we construct a *buffer chain* model as shown in the *Buffer* diagram of Figure 3.

There exist three-part buffers in the buffer chain, including host buffers, NIC buffers, and switch buffers. ① Host buffers are used to maintain messages from/to the application, as well as packets that are formed by decomposing messages (or packets that will be constructed into messages). Besides, some applications also have buffers, such as MessageQueue [45]. ② Sending side NIC buffer keeps packets delivered from the transport layer, and regularly schedules packets out into networks. Meanwhile, receiving side NIC buffer often stores packets from the network, and wait for the end-host network stack to pull packets or actively write packets into host memory. ③ Network switch buffers keep packets for switching, once the packets cannot be instantaneously forwarded, *i.e.,* a packet will be buffered or dropped in the egress queue of the port that connects the chosen next-hop.

By using different I/O techniques (*e.g.,* zero-copy), the exact number of buffers a request will experience may differ from this model. Essentially, this model provides a methodology for monitoring the end-to-end datapath of request.

3.3 Event Definition & Generation

Event definition and generation will determine the effectiveness and overhead of BufScope. BufScope designs them based on the principles of high coverage and low overhead.

Buffer classification. The buffer chain includes diverse types of buffers, which have different RLA-related abnormal events. Taking the switch buffer as an example. For lossy Ethernet, when the queuing length of a switch buffer exceeds a certain threshold, subsequent arrival packets will be dropped instead of entering the buffer, incurring a *drop* event. For lossless Ethernet, when the buffer of a downstream switch is congested, the upstream switch will pause packet forwarding until the downstream switch buffer has space for new packets, causing a *pause* event. Another example, order-sensitive buffers, such as TCP receiving buffer, may encounter head-of-line blocking (HOL), while order-insensitive buffers do not. BufScope is challenged to thoroughly analyze all types of buffers, and define the events for them respectively.

To address the challenge, we observe that though there exist various types of buffers, they could be classified according to three key properties, *i.e.,* priority awareness, order sensitivity, and enqueue feature. Priority awareness is a property for a buffer with multiple queues. If strict priority is maintained across different queues (*i.e.,* priority-aware), packets in a low priority queue will have to wait for the high priority queue to drain. Otherwise, packet dequeuing follows the FIFO principle (*i.e.,* priority-unaware). Order sensitivity refers to whether a buffer maintains strong orders of arrived packets before popping them for subsequent processing. Enqueue features are different for lossy and lossless buffers as mentioned above.

Event definition. According to the different type of the three properties, we define five kinds of buffer events which may cause RLAs, as shown in Table 1. We not only consider the occurrence of events, but also capture the detailed causes.

- *Priority contention.* This type of event is triggered in priority-aware buffer (*i.e.,* multi-level priority queue) when the lengths of higher-priority queues exceed a certain threshold, blocking the packets in low-priority queues for a long time. Inappropriate priority allocation may cause RLAs [46]. Conversely, FIFO buffers always first forward the packets that arrive earlier, and don't have this event.

Table 1: Buffer event definition. "•" means that the cause for this event happens right within this buffer, "←" means that the cause happens before this buffer (in a preceding buffer or program), and "→" means that the cause happens after this buffer.

Property	Type	Event	Triggering Condition	Cause Location	Event Information
Priority awareness	priority-unaware	-	-	-	-
	priority-aware	*priority contention*	Queuing delay exceeds a threshold & Lengths of higher-priority queues exceed a threshold	•	- Request ID - Egress queue - Lengths of higher-priority queues - Queuing delay
Order sensitivity	order-sensitive	*out-of-order*	Inconsecutive sequence number	←	- Request ID - ID of out-of-order request - Queuing delay
	order-insensitive	-	-	-	-
Enqueue feature	lossy	*drop*	Queues are about to be full or already full	•	- Request ID - Egress queue - Egress port - Ingress port
	lossless	*pause*	Receiving a PAUSE signal	→	- Request ID - Egress queue - Egress port - Queuing delay
-	-	*congestion*	Queuing delay exceeds a threshold & no PAUSE signal	•	- Request ID - Egress queue - Egress port - Queuing delay

- *Out-of-order.* This type of event is triggered in order-sensitive buffers such as TCP receiving buffer. Packets have to be delivered to the applications in the same order as they are sent. That is, the packets that have been received by the order-sensitive buffers have to be delayed before receiving the packets sent earlier. In contrast, the order-insensitive buffers don't have the out-of-order event.

- *Drop.* This event happens in lossy buffers when queues are about to be full or already full. Packet drops would incur packet retransmission and may result in RLAs.

- *Pause.* This type of event happens in lossless buffer. Once the buffer occupancy of the downstream switch exceeds a specific threshold, then the downstream switch sends a PAUSE signal to the upstream switch. The latter will pause packet forwarding until a RESUME signal is received. This increases the delay of the paused packets.

- *Congestion.* This type of event could happen to any kinds of buffers. Congestion is defined as the situation where the queuing delay exceeds a threshold, and is not due to PAUSE. This could be caused by the mismatch between upstream and downstream processing or transmission rates.

Based on that, we could predict the RLA-related events that will occur in a buffer, and deploy monitoring mechanism accordingly. Note that the events are not exclusive with each other, multiple events may be captured by BufScope simultaneously, such as congestion and priority contention.

Event generation. Event generation, which includes the event detection and collection, could degrade the monitored application performance due to its expensive operation, *e.g.,* generating unique ID, writing disk and *etc.* [10]. Thus, they must be low overhead in BufScope. Here, we describe how they are designed in the hosts, SmartNICs and switches, respectively.

In the end-host and SmartNIC, event detection in software is straightforward. In order to reduce the impact of event collection on application performance, BufScope's agent daemon executes disk write asynchronously. Then, the agent daemon sends the event logs to the *Event Collector* in bulk.

In the programmable switch, event detection needs to be implemented entirely in the data plane. Packets that experience *pause* or *drop* were detected in the ingress pipeline and MMU, respectively. For *priority contention* and *congestion*, we record the length of queues, ingress and egress timestamps through INT (in-band network telemetry) [47], and judge whether packet's queuing delay and length exceed the thresholds. After the victim packets are detected, egress agent utilizes a bloom filters to aggregate them into request-level events with flow's 5-tuple and request ID (§3.4) as the key. Then, request events are sent to the switch control plane via data plane generated packets. Finally, the control plane will eliminate false positive in received events, and report them to the *Event Collector* in bulk. Most of the design of event generation in the data plane was proposed in NetSeer [8], and we simply changed the granularity of monitoring events from flow to request. We do not claim any novelty here.

3.4 Request-level Semantic Injection

The layered network architecture [48] has been a cornerstone of the Internet and is used in clouds. However, it presents a challenge for performance monitoring of distributed applications. Specifically, these teams, *e.g.,* network (Layer-3), server (Layer-4) and application (Layer-7), often blame each other for diagnosing RLAs due to the inconsistent semantics of their monitoring tools [8, 24]. To address this challenge and improve the accuracy of RLA diagnosis, BufScope needs to
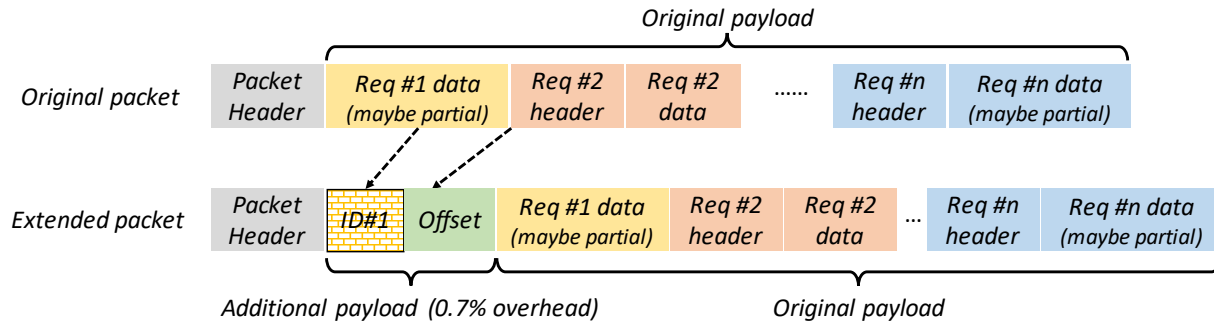
Figure 4: Request-level semantic injection in sender's NIC. (*Req* stands for request.)

map all captured events to the request-level semantics.

Challenge of request-level semantic parsing. A request contains two major parts, *i.e.,* header and payload. The header includes request ID, type (request/response), length, and other metadata, while payload holds the actual content of the request. The network stack is responsible for encapsulating the request into packets, with the request as the packet payload. In this process, multiple requests are considered as a byte stream and packed into packets. One large request could be carried by multiple (maybe >1,000 for storage applications) packets, while multiple (maybe >10) small requests may be consolidated in one packet. Therefore, each packet may not carry or carry multiple request headers in practice.

Realizing request-level semantic parsing in networks is non-trivial. The request header may be anywhere in the packet payload, which makes the commodity switch unable to parse the request IDs. A straightforward solution is inserting all IDs appeared in the packet before the packet payload, so that programmable switches could match and derive request IDs. However, this solution results in significant overhead. First, one request ID has 8 bytes in gRPC [43]. The standard MSS of TCP packets is 1460 bytes. Therefore, inserting 10 request IDs would introduce a 5% bandwidth overhead. This overhead is ever-present, and can lead to bandwidth degradation and packet loss under traffic bursts. Furthermore, performing lots of memory copy operations seriously wastes CPU resources.

Concise request-level semantic injection. To address this challenge, we leverage the fact that request *ID* and *length* are already carried in the request headers packed in packet payloads. Thus, we choose to insert the *offset* field (2 bytes) of the first complete request header at the beginning of the packet payload, as shown in Figure 4. If there are other request headers, we can use the *length* field in their headers to iteratively parse their indexes. By performing such concise operation, we explicitly maintain the request-level information in a way which programmable switches (*e.g.,* P4-16 [49]) could easily parse, while introducing very little additional overhead on performance and bandwidth.

Besides, the first data segment of the payload, *i.e.,* the partial *Req#1 data* in Figure 4, may not has the corresponding header in the current packet, because large request could be carried by multiple packets. Therefore, we should also main-

---

Algorithm 1: Semantic injection in sender's NIC

Input: *Packet*, *last_ID*

1 $ID\#1\_index \leftarrow tcp\_payload\_begin$;
2 $offset\_index \leftarrow tcp\_payload\_begin + ID\_len$;
3 $insert\_len \leftarrow ID\_len + offset\_len$;
4 buf_append(*Packet*, *insert_len*);
5 buf_move($ID\#1\_index + insert\_len, ID\#1\_index$);
6 **while** $index + + < tcp\_end$ **do**
7   **if** $*index = Request.header$ **then**
8   **if** $now\_ID = NULL$ **then**
9   **if** $index = ID\#1\_index + insert\_len$ **then**
10             $ID\#1\_index \leftarrow NULL$;
11   **else**
12             $*ID\#1\_index \leftarrow last\_ID$;
13          $*offset\_index \leftarrow index$;
14       $*now\_ID \leftarrow *index$;

15 **if** $now\_ID = NULL$ **then**
16    $*ID\#1\_index \leftarrow last\_ID$;
17    $offset\_index \leftarrow NULL$;
18 **else**
19    $last\_ID \leftarrow *now\_ID$;

---

tain the identifier of the *Req#1 data*. To this end, we always insert the *ID* field (8 bytes) at the beginning of the packet payload, which records the *Req#1 ID*. It requires us to maintain a stateful variable, which records the ID of the recent request. The bandwidth overhead of our injection solution is only 0.7% ($(8+2)/1460$), which is fixed and negligible.

SmartNIC-offloaded semantic injection and recovery. To reduce the CPU overhead in hosts, we offload the semantic injection to SmartNICs. We present the process of semantic injection in Algorithm 1. For each packet, we first insert a fixed space to store the *offset* and *ID field* (line 1-5). Then we look at three possible scenarios. ① These is no partial *Req#1 data* and there is a complete *Req#2 header* at the beginning of the payload (line 9-10); ② These is partial *Req#1 data* and a complete *Req#2 header* (line 11-14); ③ There are no request headers in this packet (line 15-17), then the stateful variable about the recent request ID does not need to be updated. Otherwise, in the first two scenarios, the variable

---

records the last request ID in the current packet (line 18-19), which may becomes the recent request ID for the next packets.

We recover the packets in the receiver's SmartNIC, which just removes the content that was inserted in the sender's SmartNIC. Through SmartNIC offloading instead of host CPU processing, semantic injection and recovery can be achieved with little impact on the application performance.

3.5 RLA Diagnosis and Mitigation

We introduce how cloud providers diagnose the cause of RLAs according to the events captured by BufScope. First, BufScope correlates events by request ID, and reports the beginning and ending events (possibly guilty) to the operators. Then, since the blocked time is also recorded in the events, operators can clearly see which event is the culprit, even through a request experiences multiple events. We also elaborate on how applications can benefit from these request events.

- *Priority contention.* This type of events suggests that RLAs happen directly in the current buffer. It indicates that queues with higher priorities are jammed. To mitigate this type of RLA, application owners can either upgrade the priority of its requests, or try to reduce the traffic of other applications that enter the high-priority queues.
- *Out-of-order.* This type of event suggests that packets are dropped or detoured in previous buffers or logic. For instance, network packet drop and path change could cause out-of-order in TCP receiving buffer. Application owners could ask network operators for help to debug network device failures, blackholes, or random packet drops. Also, refer to the next item if accompanied by drop events.
- *Drop.* Drop events cause time-consuming retransmission, which could directly cause RLAs. MMU drop is usually caused by burst and incast. Application owners could consider optimizing the traffic pattern through scheduling to reduce TCP incast or congestion possibilities.
- *Pause & Congestion.* These events happen due to the slow scheduling of packets out of the current buffer or the downstream buffer. In this case, BufScope could identify the request that contribute the most to the congestion, *i.e.,* the heavy request, because the heavy request will experiences more congestion events. Then, cloud providers need to evaluate the network architecture and application mixing model.

4 Implementation

We have implemented BufScope for a kernel-based RPC framework named Finagle [50] and the kernel-bypass-based *Alibaba*'s block storage application. We use Barefoot Tofino switches and Broadcom PS225 SmartNICs to implement the functions of BufScope in the data plane.

Requirements. Because BufScope needs to insert the ID and offset field of the request at the end of the packet header, implementation of BufScope requires application-layer protocol awareness and MTU modification. And kernel-intrusive is needed for monitoring kernel-based application. Finally, BufScope is designed to monitor requests inside the cloud (*i.e.,* east-west traffic) that are typically not encrypted.

Incremental deployment. For end-to-end monitoring, multiple teams (*e.g.,* server and network) in the cloud need to monitor the buffers they manage by using BufScope's APIs. However, partial deployment of BufScope still facilitates RLAs diagnosis. With sole support from the server team, semantic injection and in-server event monitoring can still be performed, which helps operators decide whether the root cause locates in the server or not. With sole support from the network team, operators can blame or exonerate the network according to packet- or flow-level events in the network.

Buffer identification. BufScope summarizes a basic buffer chain for various applications and network stacks. For kernel-based, there is an application buffer and a socket buffer. For kernel-bypass-based, the zero-copy technology makes the applications may have only one *mbuf* array for DPDK [51]. The manual efforts to identify buffers are small and only need to be done once. Moreover, resource contention in other hardware or OS queues (*e.g.,* CPU, DRAM, and PCIe) will cause slow message processing, resulting in queue buildup in the upstream buffer [52], which can be detected by BufScope.

Event capturing. We record the following necessary information for each type of events.

- *Priority contention* (15B): *<ID, egress queue, length of higher-priority queues, queuing delay>*. We measure the queuing delay inside a switch with ingress and egress timestamps. The victim request ID, the timestamps and the length of the higher-priority queue is obtained by INT.
- *Out-of-order* (20B): *<ID, ID of out-of-order request, queuing delay>*. We identify out-of-order requests by observing inconsecutive sequence number in packets, and generate this type of events for latter blocked requests.
- *Drop* (11B): *<ID, egress queue, egress port, ingress port>*. In network, we redirect packets dropped by MMU to a dedicated internal port, and report in egress pipeline [8], then parse the request ID in these packets.
- *Pause* (14B): *<ID, egress queue, egress port, queuing delay>*. For a lossless network, the switch begins to generate pause events immediately after receiving the pause signal.
- *Congestion* (14B): *<ID, egress queue, egress port, queuing delay>*. Congestion events are produced when the queuing delay exceeds a threshold while the length of the higher-priority queue is normal.

To reduce the bandwidth overhead, we leverage lossless *ZigZag Encoding* [53] to compress events. The average length of events is shortened from 15 bytes to 8 bytes. Besides, BufScope allows setting an upper limit on the event generation rate. Once this threshold is exceeded, sampling can be enabled.

SmartNIC. We implement the NIC buffer monitoring and semantic injection in the ARM-based SmartNIC. In addition to the cores required for packet forwarding, BufScope

requires only one additional core for event collection and reporting. Note that the ID of the partial *Req#1 data* in retransmitted packets has been missed in NIC. We just set the inserted *ID field* as *NULL* in the retransmitted packets. The effects of this simplification are limited, because the causes of packet retransmission have already been captured (*e.g.,* drop or out-of-order). When enabling TSO, semantic injection is performed after packets are segmented in SmartNIC.

Switch. BufScope's ASIC logic can be embedded into original switch programs (*switch.p4* in our experiment) as an extension. We layout the timestamp record and pause detection modules in ingress pipelines, enable drop detection in the MMU, and detect congestion, priority contention in the egress pipeline. After the event is generated by the switch ASIC, event pre-processing and reporting in the switch CPU is similar to that in the NetSeer system [8].

Event collector. To timely receive events, we use the servers with 100Gbps NICs in the cluster as the *Event Collector*. To improve the readability and usability of monitored data, Collector aggregate the events in two stages. First, the events captured by all components are aggregated together at per-request granularity. Then, if there is a trace data generated by the tracing tool for the request, the request events timestamp and the associated information are marked in that trace.

5 Evaluation

Environment: We evaluate BufScope and existing monitoring tools on a testbed with a 4-ary and 3-tier Fat-Tree topology [54] composed of 10 Barefoot Tofino switches and 16 servers. Each server has 192 CPU cores, 64GB RAM, and one Broadcom PS225 SmartNICs (2×25G) [55]. Each SmartNIC possesses eight ARM Cortex-A72 3.0 GHz CPUs and 16GB memory. There are 4 ToR, 4 Aggregate and 2 Core switches. They are interconnected with 100G links, while each ToR connects four servers with 2×25G link.

Baselines: Given that none of the existing monitoring tools are designed to monitor the end-to-end datapath of request, we combine multiple state-of-the-art tools to fully cover it:

(i) Application tracing. We enable an open-source tracing tool [15] to capture the RPC timing data and application-specific abnormal events in application layer. Because of the large amount of captured data and non-negligible CPU overhead, tracing tools typically require sampling (*e.g.,* 0.1% under high-load services [10]) to reduce impact on the performance of the monitored application. Thus, we set the sampling rate of tracing as 0.1% and 100% for comparison.

(ii) TCP monitoring. Dapper [21] is used to diagnose performance problem of TCP in the end-host network stack.

(iii) Network monitoring. We deploy NetSeer [8] and packet sampling to capture events in networks and NICs. NetSeer is a flow-level event monitoring system based on programmable data plane. For packet sampling, we can parse the request ID in the mirrored packet offline to get request events. Since it
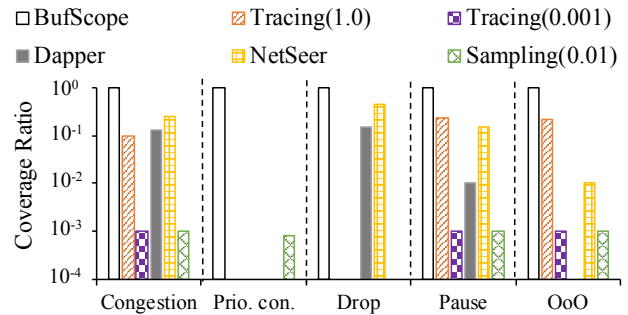
Figure 5: Event coverage ratios. Tracing(sampling rate).

has a large bandwidth overhead under a high sampling rate [8], we configure the sampling rate as 1%.

The evaluation needs to answer the following 3 questions.

- **Coverage:** Can BufScope capture most (close to 100%) request events that happen in hosts, NICs, and network switches, and help accurately diagnose the real RLAs?
- **Scalability:** What's the bandwidth overhead of BufScope to deliver events to the *Event Collector*? Can it scale with the increasing datacenter size and bandwidth?
- **Performance overhead:** How to choose an efficient threshold? How does BufScope affect the application performance? How about the impact of each module, including request-level semantic injection by host CPU or SmartNIC?

5.1 Coverage

We deploy the storage application (supports RDMA) and Finagle as the monitored applications, and run traffic traces based on four real-world workloads including DCTCP [56], VL2 [57], storage and WEB [58] for 6 hours. We set the average link utilization as 80% to test BufScope's coverage under extreme conditions. *Congestion*, *drop* and *out-of-order* (OoO) are naturally produced. We configure various priority queues to trigger *priority contention*, and enable priority flow control (PFC [59]) in RDMA network to trigger *pause*, which do occur in production environments [8, 37]. We start by evaluating BufScope's capability to fully capture all events along the datapath of request. Next, we compare the proportion of unexplained RLAs of different monitoring tools, and study 2 real RLAs which cause the SLA violations.

Event coverage. We enable tracing, Dapper, NetSeer, packet sampling and BufScope to capture events, respectively. We present the event coverage ratios for different types of events in Figure 5. For a fair comparison, we enable tracing tool in this experiment to monitor all buffer events in host buffers that it can cover. Even so, the tracing tool only has visibility into applications, it cannot detect events in the network. Therefore, tracing(1.0) can only cover up to 23% events, while tracing(0.001) can only cover 0.1% events. Dapper analyzes TCP statistics to infer the occurrence of network events, such as congestion and drop, it can only cover up to 15% events.

NetSeer could capture flow-level events in networks, including congestion, pause and drop, but leaves out the priority
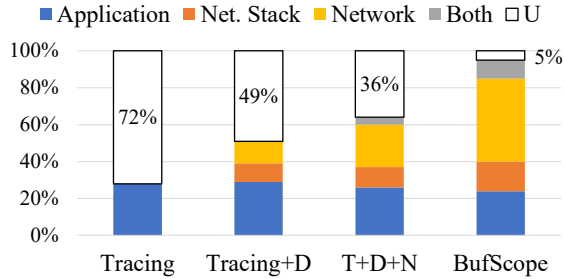
Figure 6: Diagnosing RLAs by different monitoring tools (T:Tracing, D:Dapper, N:NetSeer, U:Unknown).

Figure 7: Bandwidth overhead of event collection.

contention and OoO events. Besides, its captured events miss request-level semantics. Based on the time-correlation methods, NetSeer can only cover up to 45% request events. For packet sampling, if the mirrored packet is lucky enough to encounter an event, then we can parse out the event with request ID. Thus, it only cover <10% events which is always less than its sampling rate. In comparison, BufScope has full coverage for the 5 types of request events happened in both of the end-hosts and networks.

Diagnosing RLAs. We try to diagnose the root cause of the slow RPC (*i.e.,* RLAs) detected during that period according to different monitoring tools. Request-level timing data collected by tracing tool with full sampling can only explain 28% RLAs, leaving 72% RLAs undetermined, as shown in Figure 6. Then, server and network monitors capture flow-level events to diagnose RLAs based on the time-correlation methods, can only explain 23% and 13% more RLAs, respectively. Even so, enabling tracing, Dapper and NetSeer (*i.e.,* T+D+N) at the same time still leaves 36% RLAs inexplicable. With BufScope's help, we can tell whether and how much each component is responsible for each slow RPC, and explain much more (95%) RLAs, including those whose causes were unknown with existing monitors, and some RLAs that were caused by multiple components. The remaining 5% of the RLAs did not reveal any events. We speculate that they are caused by hardware-related anomalies.

We reproduce 2 real *Alibaba*'s production RLAs on our testbed with inferred topology, requests pattern, and traffic rate during the incidents, which cannot be captured and explained by existing monitoring tools.

#1) Polling hang in the host. When a request encounters more than one anomaly, the challenge in diagnosing is to identify the one that has the greatest impact. For example, one RPC in this experiment encountered congestion in the network and polling hang in the receiver. However, existing monitoring tools cannot capture how much delay each anomaly causes, and treat them equally, resulting in the inefficient diagnostic process. Conversely, BufScope can end-to-end capture latency-critical events with consistent semantics, which can associate events that occur in different components. BufScope found that it was blocked for 5*ms* at the receiver's NIC, and only experienced 80*μs* of congestion in the network. Therefore, the reason for the RLA was polling hang. Based on this,
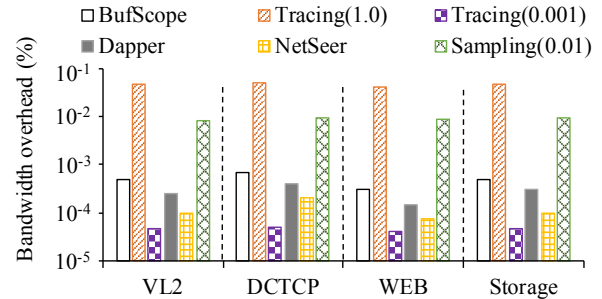
application owners can further analyze the abnormal events in the host and the system log to solve the problem.

#2) Cascaded priority contention. In a priority-aware network, it is non-trivial to assign priorities to different applications. An inappropriate allocation can result in SLA miss for low-priority application. Thus, checking the priority contention in the network is an important task of performance monitoring tools. Worse still, a cascading effect would happen when there are multiple queues with diverse priorities. Consider there are three requests, *Req#1*, *Req#2*, and *Req#3* have decreasing order of priority. *Req#1* and *Req#2* contend in an upstream switch S_1, while *Req#2* and *Req#3* contend in a downstream switch S_2. If *Req#1* in S_1 is congested, *Req#2* would be delayed, which then delays *Req#3* in the low-priority queue of S_2. To debug the RLAs of *Req#3*, simply observing the priority contention in a switch is not enough. Since BufScope can capture all contention events and their details, we can analyze this cascade effect and find a more effective method to mitigate it.

5.2 Scalability

We compared the bandwidth overhead (BO) required by BufScope and baselines to report events or traces during that period. Figure 7 shows that BufScope only incurs <0.07% BO under various real-world workloads, of which 0.02% from host, 0.01% from NICs and 0.04% from switches. For link bandwidth at 100Gbps, the overhead is at most 70Mbps, which is within the capacity of PCIe (18Gbps) and switch CPU (13.4Gbps with 2 cores). In comparison, tracing(1.0) suffers from >4% BO, its each span (*i.e.,* every request) needs 400B on average. Tracing(0.001) needs >0.004% BO. Dapper records only TCP abnormal events and consumes only 0.04% BO. Network packet sampling (0.01) needs ∼ 1% BO, which is similar to its sampling rate, because the payload also needs to be recorded to parse request semantics. Because NetSeer captures and reports flow-level events, its event scale and fineness are not as high as BufScope. Thus, NetSeer only incurs ∼ 0.01% BO. In summary, T+D+N consumes >4.05% BO.

To further understand the scalability of BufScope, we calculate the monitoring event traffic as well as the processing overhead of BufScope according to the configuration of *Alibaba*'s production datacenters. For a normal 3-tier datacenter, connecting 10,000 servers requires approxi-
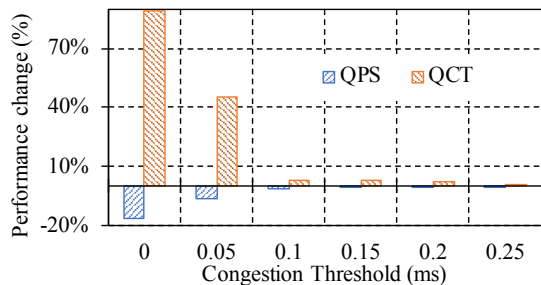
Figure 8: Impact of congestion threshold on performance.
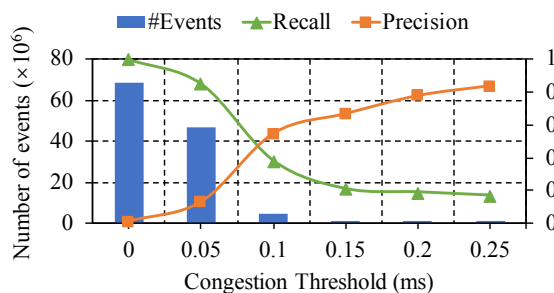
Figure 9: Number of captured congestion events and the two indicators under different thresholds.

mately 400 switches (3.2Tbps), which produce a maximum of $400 \times 3200Gbps \times 0.07\% = 896Gbps$ monitoring traffic at most. Processing such traffic requires 9 servers with 100Gbps NICs, which implies a 0.09% processing overhead.

5.3 Performance Overhead

In this experiment, we use as many threads as possible, which perform 4KB file read, to test the extreme performance of the *Alibaba*'s storage application under different monitoring tools. We first show a method for selecting the appropriate congestion event threshold. Then, we evaluate the performance overhead of the per-module and overall BufScope.

Congestion event threshold determination. Congestion events appear when the queuing delay exceeds a certain threshold, which we define as the congestion threshold. We pay special attention to congestion events as it occupies a major portion of all events, which will occur in both the hosts and networks. BufScope can harm application performance if too many congestion events are collected. Thus, we run the application for 10 seconds with a full-mesh traffic pattern, and measure the impact on the QPS (Query per Second) and QCT (Query Completion Time) of the application as we vary the congestion threshold. As shown in Figure 8, the larger the threshold, the smaller the performance overhead, because fewer congestion events would be collected. Thus, threshold selection is highly related to the efficiency of BufScope.

Since requests without RLA can also experience light congestion, this means that not all captured congestion events are RLA-related. In this experiment, the captured events are RLA-related when the RLA request experiences only congestion events and no other events. We use two indicators to
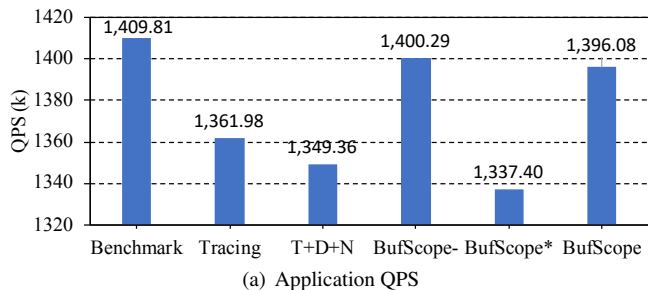
(a) Application QPS
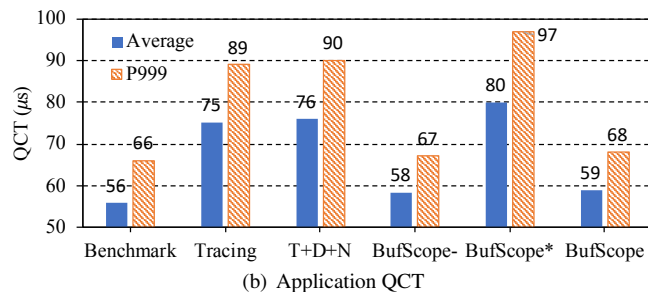
(b) Application QCT

Figure 10: Application performance under different tools.

evaluate the efficiency of the congestion threshold. Recall represents the proportion of the captured RLA-related congestion events in all real RLA-related congestion events, while precision represents the proportion of the captured RLA-related congestion events in all captured congestion events. The higher the threshold, the lower the recall and the higher the precision. Thus, the selection of the threshold needs to balance these two indicators. Figure 9 shows the changes in the number of events collected, the recall and precision. We observe that as the threshold increases from 0 to 0.25ms, the precision increases from a very low value to nearly 100%, and recall drops from 100% to a very low value. In the following experiments, we take 0.1ms as the congestion threshold for high monitoring efficiency.

Event monitoring in hosts. Because monitoring in hosts uses expensive CPU resources, we evaluate the performance overhead of only enabling functions of BufScope in hosts (we refer to this variation as BufScope-). As shown in Figure 10, we generate the highest load (*i.e.,* extreme throughput of NIC) with 8 threads to test the application, and obtain the QPS, average and P999 QCT by running 30 seconds. The Benchmark represents the raw performance of the application without any monitoring tools. BufScope- decreases the QPS by 0.7% and increases the P999 QCT by 1.5%. Because BufScope records events asynchronously, most of this overhead comes from event detection, which takes tens of nanoseconds on average. In contrast, the tracing tool generates a trace for each sampled request, which takes sub-microseconds. Thus, only enabling tracing(1.0) in hosts decreases the QPS by 3.4% and increases the P999 QCT by 34.8% under the same load. This demonstrates that BufScope's event-driven approach significantly reduces the performance overhead.

Semantic injection in network stack. Next, we enable all monitoring functions of BufScope in hosts, SmartNICs and

switches. In this experiment, we evaluate the impact on the RTC application by implementing the BufScope's semantic injection in the application's network stack, namely BufScope*. As shown in Figure 10, since the semantic injection uses the same thread with the application processing, BufScope* decreases the QPS by 5.1% and increases the P999 QCT by 47.0%. In comparison, the combination of tracing, Dapper and NetSeer, *i.e.,* T+D+N, decreases the QPS by 4.3% and increases the P999 QCT by 36.4%. BufScope*'s performance overhead is large than the combination. The results reveal that the overhead of using the same CPU to perform semantic injection is not negligible, and we need to use offload techniques to reduce the overhead.

Overall performance overhead of BufScope. According to BufScope's design, here semantic injection is implemented in the sender's SmartNIC. BufScope only decreases the QPS by 1.0% and increases the P999 QCT by 3.0%. This demonstrates that *SmartNIC-offloaded semantic injection and recovery* can significantly reduce the performance overhead. Compared with T+D+N, BufScope improves the QPS by 3.5% and reduces the P999 QCT by 24.4%. Besides, the performance of BufScope is slightly lower than that of BufScope-. Such performance decline is introduced by our ARM-based implementation in SmartNIC. We will use FPGA-based SmartNIC in the future to further improve processing performance.

6 Related Work

There has been a rich literature regarding application monitoring and diagnosis. We classify them into six categories according to their coverage for the request's datapath.

Tracing-based. Tracing-based monitoring tools are widely used for large-scale application performance tracing and debugging [9–14,16,28,30,60,61]. By inserting annotations into the execution path of the request, tracing tools can locate the problematic step in application layer, but has no visibility in the network stack and underlying networks. Besides, tracing could provide fine-grained latency statistics, but will actually degrade application performance [10]. Therefore, tracing are often used in an on-demand and sampling way.

Log-based. Log analysis is proven effective in many programs or performance debugging scenarios [3, 29, 62–67]. However, logs are often created by CPU, which is proven inefficient and could waste much CPU resources. Therefore, log-based monitoring systems often use second-level monitoring granularity, which will miss a lot of RLAs.

Network stack-based. Many researches are trying to monitor network performance on the end-host network stack. For example, some research efforts propose to constantly monitor TCP performance by watching TCP statistics such as timeout and retransmission, and deduce the root cause of RLAs through statistical analytics [17, 21, 31, 68], replay [25], or machine learning [24]. Trumpet [23] leverages triggers at end-hosts to monitor every packet and network-wide events.

However, they lack visibility into the network, leading to the incomplete coverage for RLAs. Moreover, they focus on packet- or flow-level event capturing and analysis, and cannot correlate events to the corresponding requests.

NIC-based. Simon [35] collects statistics from NICs, and reconstruct flow queuing time, link utilization, link composition, and other statistics. Nevertheless, it could only obtain aggregated statistics with millisecond-level granularity and lose clues for events at fine-timescale such as microsecond-level microbursts. Similarly, it mainly focuses on network events and cannot fully detect host events.

Network-based. The network serves as the conjunction component among distributed servers. Thus, many efforts have been devoted to network monitoring by active probing [19, 22], telemetry [18, 32], *etc.* NetSeer [8] leverages programmable switch to monitor flow-level network abnormal events, without the request-level semantics. Retro [33] and Microscope [26] monitor the queue to identify anomalies, which is similar to BufScope's buffer model. However, network-based monitoring tools have no visibility into hosts. Moreover, their combination with tracing cannot improve the accuracy of RLAs diagnosis due to the inconsistent semantics.

Network and host collaboration. Recent researches use both the network and end-host to jointly collect, store and analyze data [34, 36–38]. In order to correlate packets' behaviour in end-hosts and networks, they often enable network switches to attach metadata to packets, and extract event in hosts, which will consume a lot of host CPU resources. Besides, these systems did not consider the request-level abnormal events and RLA diagnosis.

7 Conclusion

This paper presents a promising way to utilize the programmable data plane to achieve high coverage for request monitoring and accurate RLA diagnosis, by proposing BufScope. Its core idea is to uniformly model the data plane in networks and the datapath in hosts using buffer. It translate most RLAs to buffer-related events, and monitor them in the buffer chain with consistent request-level semantic. Testbed-based evaluations show that BufScope can diagnose 95% RLAs with negligible bandwidth and performance overhead.

Acknowledgement

References

[1] CNCF. Cloud native computing foundation: https://cncf.io/, 2021.

[2] Patrick Stuedi, Animesh Trivedi, Bernard Metzler, and Jonas Pfefferle. Darpc: Data center rpc. In *ACM SoCC*, 2014.

[3] Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan Ullah, Yu Luo, Ding Yuan, and Michael Stumm. lprof: A non-intrusive request flow profiler for distributed systems. In *USENIX OSDI*, 2014.

[4] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed data structures over a shared log. In *ACM SOSP*, 2013.

[5] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, et al. The ramcloud storage system. In *ACM TOCS*, 2015.

[6] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter rpcs can be general and fast. In *USENIX NSDI*, 2019.

[7] Yixiao Gao, Qiang Li, et al. When cloud storage meets RDMA. In *USENIX NSDI*, 2021.

[8] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, et al. Flow event telemetry on programmable data plane. In *ACM SIGCOMM*, 2020.

[9] Rodrigo Fonseca, George Porter, Randy H Katz, and Scott Shenker. X-trace: A pervasive network tracing framework. In *USENIX NSDI*, 2007.

[10] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. 2010.

[11] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, et al. Canopy: An end-to-end performance tracing and analysis system. In *ACM SOSP*, 2017.

[12] Arjun Satish, Thomas Shiou, Chuck Zhang, Khaled Elmeleegy, and Willy Zwaenepoel. Scrub: online troubleshooting for large mission-critical applications. In *ACM EuroSys*, 2018.

[13] Dan Ardelean, Amer Diwan, and Chandra Erdman. Performance analysis of cloud applications. In *USENIX NSDI*, 2018.

[14] Uber Technologies. Jaeger: open source, end-to-end distributed tracing. https://www.jaegertracing.io/, 2020.

[15] Twitter. Zipkin. http://zipkin.io/, 2021.

[16] CNCF. Opentelemetry. http://opentelemetry.io/, 2021.

[17] Minlan Yu, Albert G Greenberg, David A Maltz, Jennifer Rexford, Lihua Yuan, Srikanth Kandula, and Changhoon Kim. Profiling network performance for multi-tier data center applications. In *USENIX NSDI*, 2011.

[18] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y Zhao, et al. Packet-level telemetry in large datacenter networks. In *ACM SIGCOMM*, 2015.

[19] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *ACM SIGCOMM*, 2015.

[20] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Lossradar: Fast detection of lost packets in data center networks. In *ACM CoNEXT*, 2016.

[21] Mojgan Ghasemi, Theophilus Benson, and Jennifer Rexford. Dapper: Data plane performance diagnosis of tcp. In *ACM SOSR*, 2017.

[22] Yanghua Peng, Ji Yang, Chuan Wu, Chuanxiong Guo, Chengchen Hu, and Zongpeng Li. detector: a topology-aware monitoring system for data center networks. In *USENIX ATC*, 2017.

[23] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Trumpet: Timely and precise triggers in data centers. In *ACM SIGCOMM*, 2016.

[24] Behnaz Arzani, Selim Ciraci, Boon Thau Loo, Assaf Schuster, and Geoff Outhred. Taking the blame game out of data centers operations with netpoirot. In *ACM SIGCOMM*, 2016.

[25] Yuliang Li, Rui Miao, Mohammad Alizadeh, and Minlan Yu. Deter: Deterministic {TCP} replay for performance diagnosis. In *USENIX NSDI*, 2019.

[26] Junzhi Gong, Yuliang Li, Bilal Anwer, Aman Shaikh, and Minlan Yu. Microscope: Queue-based performance diagnosis for network functions. In *ACM SIGCOMM*, 2020.

[27] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. Tcp = rdma: Cpu-efficient remote storage access with i10. In *USENIX NSDI*, 2020.

[28] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using magpie for request extraction and workload modelling. In *USENIX OSDI*, 2004.

[29] Yongle Zhang, Serguei Makarov, Xiang Ren, David Lion, and Ding Yuan. Pensieve: Non-intrusive failure reproduction for distributed systems using the event chaining approach. In *ACM SOSP*, 2017.

[30] Yongle Zhang, Kirk Rodrigues, Yu Luo, Michael Stumm, and Ding Yuan. The inflection point hypothesis: a principled debugging approach for locating the root cause of a failure. In *ACM SOSP*, 2019.

[31] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, and Alex C Snoeren. Passive realtime datacenter fault detection and localization. In *USENIX NSDI*, 2017.

[32] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *USENIX NSDI*, 2014.

[33] Jonathan Mace, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. Retro: Targeted resource management in multi-tenant distributed systems. In *USENIX NSDI*, 2015.

[34] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. Confluo: Distributed monitoring and diagnosis stack for high-speed networks. In *USENIX NSDI*, 2019.

[35] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. Simon: A simple and scalable method for sensing, inference and measurement in data center networks. In *USENIX NSDI*, 2019.

[36] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Simplifying datacenter network debugging with pathdump. In *USENIX OSDI*, 2016.

[37] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Distributed network monitoring and debugging with switchpointer. In *USENIX NSDI*, 2018.

[38] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray failure: The achilles' heel of cloud-scale systems. In *USENIX HotOS*, 2017.

[39] Arnaldo Carvalho De Melo. The new linux 'perf' tools. In *Slides from Linux Kongress*, volume 18, 2010.

[40] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. Acceltcp: Accelerating network applications with stateful {TCP} offloading. In *USENIX NSDI*, 2020.

[41] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling programmable transport protocols in high-speed nics. In *USENIX NSDI*, 2020.

[42] Apache. Thrift. http://thrift.apache.org/, 2020.

[43] Google. grpc: A high-performance, open source universal rpc framework. https://grpc.io/, 2020.

[44] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mtcp: a highly scalable user-level {TCP} stack for multicore systems. In *USENIX NSDI*, 2014.

[45] Apache. Rocketmq. https://rocketmq.apache.org/, 2021.

[46] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, Ori Rottenstreich, Steven A Monetti, and Tzuu-Yi Wang. Fine-grained queue measurement in the data plane. In *CoNEXT*. ACM, 2019.

[47] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. In-band network telemetry via programmable dataplanes. In *ACM SIGCOMM*, 2015.

[48] I Standardization. Iso/iec 7498-1: 1994 information technology–open systems interconnection–basic reference model: The basic model. *International Standard ISOIEC*, 74981:59, 1996.

[49] Mihai Budiu and Chris Dodd. The p416 programming language. *ACM SOSP*, 2017.

[50] Twitter. Finagle. http://twitter.github.io/finagle/, 2021.

[51] DPDK Intel. Data plane development kit. http://dpdk.org, 2014.

[52] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan MG Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, et al. Swift: Delay is simple and effective for congestion control in the datacenter. In *ACM SIGCOMM*, 2020.

[53] Google. Protocol buffers: Encoding: Signed integers. https://developers.google.com/protocol-buffers/docs/encoding#signed_integers, 2021.

[54] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *SIGCOMM computer communication review*, 2008.

[55] Broadcom. Stingray ps225 smartnic. https://www.broadcom.com/products/ethernet-connectivity/smartnic/ps225, 2020.

[56] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *ACM SIGCOMM*, 2010.

[57] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: a scalable and flexible data center network. In *ACM SIGCOMM*, 2009.

[58] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. Inside the social network's (data-center) network. In *SIGCOMM*, 2015.

[59] Ieee standard for local and metropolitan area networks–media access control (mac) bridges and virtual bridged local area networks–amendment 17: Priority-based flow control. *IEEE Std 802.1Qbb-2011 (Amendment to IEEE Std 802.1Q-2011 as amended by IEEE Std 802.1Qbe-2011 and IEEE Std 802.1Qbc-2011)*, pages 1–40, 2011.

[60] Úlfar Erlingsson, Marcus Peinado, Simon Peter, Mihai Budiu, and Gloria Mainar-Ruiz. Fay: extensible distributed tracing from kernels to clusters. In *ACM TOCS*, 2012.

[61] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. In *ACM TOCS*, 2018.

[62] Xu Zhao, Kirk Rodrigues, Yu Luo, Ding Yuan, and Michael Stumm. Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle. In *USENIX OSDI*, 2016.

[63] Liang Luo, Suman Nath, Lenin Ravindranath Sivalingam, Madan Musuvathi, and Luis Ceze. Troubleshooting transiently-recurring errors in production systems with blame-proportional logging. In *USENIX ATC*, 2018.

[64] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. Be conservative: enhancing failure diagnosis with proactive logging. In *USENIX OSDI*, 2012.

[65] Mike Y Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *IEEE DSN*, 2002.

[66] Karthik Nagaraj, Charles Killian, and Jennifer Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *USENIX NSDI*, 2012.

[67] Srikanth Kandula, Ratul Mahajan, Patrick Verkaik, Sharad Agarwal, Jitendra Padhye, and Paramvir Bahl. Detailed diagnosis in enterprise networks. In *ACM SIGCOMM*, 2009.

[68] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Boon Thau Loo, and Geoff Outhred. 007: Democratically finding the cause of packet drops. In *USENIX NSDI*, 2018.

Characterizing Physical-Layer Transmission Errors in Cable Broadband Networks

Jiyao Hu
Duke University

Zhenyu Zhou*
Duke University

Xiaowei Yang
Duke University

Abstract

Packet loss rate in a broadband network is an important quality of service metric. Previous work that characterizes broadband performance does not separate packet loss caused by physical layer transmission errors from that caused by congestion. In this work, we investigate the physical layer transmission errors using data provided by a regional cable ISP. The data were collected from 77K+ devices that spread across 394 hybrid-fiber-coaxial (HFC) network segments during a 16-month period. We present a number of findings that are relevant to network operations and network research. We estimate that physical-layer errors can contribute to 12% to 25% of packet loss in the cable ISPs measured by the FCC's Measuring Broadband America project. The average error loss rates of different HFC network segments vary by more than *six* orders of magnitude, from $O(10^{-6}\%)$ to $O(1\%)$. Users in persistently high-error-rate networks do not report more trouble tickets than other users.

1   Introduction

Reliable and high-speed Internet access is increasingly important to modern life, especially in a pandemic. According to [7], the number of broadband subscribers in the U.S. exceeded 105 million by the end of 2020. The availability and quality of broadband networks are of great policy concerns, as the U.S. government seeks to ensure an affordable and high-quality Internet service is provided to all [1].

In 2011, the Federal Communications Commission (FCC) launched the Measuring Broadband America (MBA) project to gain insight into the operational conditions of broadband networks [2]. The MBA project enlisted thousands of volunteers residing in ten U.S. ISPs and installed customized devices inside their homes. These devices send continuous measurement packets to estimate performance metrics such as packet loss rates, round trip latencies, and download/upload speeds of the volunteers' broadband networks. Similarly, much previous work measured and characterized different aspects of the last-mile broadband access

networks, including latency, loss, throughput, and availability [8, 17, 18, 25, 26, 28–30].

FCC's MBA project and previous work provide useful insight into how U.S. broadband networks perform. Among the metrics they gather, the packet loss rate is a particularly important Quality of Service (QoS) metric, as it affects TCP throughput as well as applications such as VoIP, live streaming, or multi-player online games. The communication quality of VoIP will significantly drop when the packet loss rate exceeds 1% [2]. In addition, the default TCP variant used by dominant operating systems, TCP Cubic [20], will reduce its sending rate after a packet loss.

However, the packet loss rates previous work measured have a severe limitation: they are end-to-end packet loss rates and do not separate the last-mile physical layer loss from other sources of packet loss. Packet loss comes from two main sources: error loss caused by the physical layer transmission errors and congestion loss caused by buffer contention at routers or switches. It is important to separate these two sources of packet loss for the following reasons.

First, physical layer packet loss is a direct indicator of how physical layer infrastructure functions, while other metrics, including latency, throughput, and end-to-end packet loss rates, are affected by multiple factors such as network capacity provisioning and router buffer management. Thus, physical layer loss can serve as a simple anomaly detector to network maintenance teams, while other metrics cannot.

Second, it is of great policy interest to monitor physical layer loss, as it is related to how well a broadband network is maintained. Broadband services in the U.S., while typically operated on existing telecommunications infrastructure (i.e., telephone or cable TV), are declassified from common carrier services [15]. Yet broadband Internet connections are increasingly becoming a public utility. Through continuous monitoring, policymakers can gauge how well the infrastructure is maintained without regulation and may consider appropriate policy adjustments if an unregulated broadband market leads to decreased quality of service.

Finally, understanding how much physical layer errors con-

---

*Zhenyu Zhou is now at Google.

tribute to end-to-end packet loss offers valuable insight into the design of congestion control algorithms and network simulations. A number of TCP variants, including Cubic [20], consider packet loss as a congestion signal. If physical layer error loss is common, we need to reexamine this assumption and possibly move away from such protocols to a more loss-agnostic one such as TCP BBR [14]. In addition, the design of a network protocol often uses simulations to evaluate the initial design. To conduct simulations, the designer often needs to configure a link's packet loss rate. To date, we do not have a clear understanding of how to configure the physical layer loss rate of a broadband link, but broadband networks are widely used in end-to-end connections. If we can separate the physical layer loss from the end-to-end packet loss, we can gain insight into how to build a physical layer error model and use it to conduct high-fidelity network simulations.

In this work, we aim to characterize packet loss caused by physical layer transmission errors. A regional cable ISP in the U.S. provides us physical layer performance data collected from 77K+ devices (primarily cable modems) every four hours from two disjoint geographical areas in a 16-month period.[2] The devices span across 394 hybrid-fiber-coaxial (HFC) network segments. Following operational practice, we refer to each HFC network segment as a fiber node (FN), as such a network segment terminates at a fiber optic node. The data we obtain include the number of unerrored, corrected, and uncorrectable DOCSIS [12] codewords a device sends since its last reboot. We develop techniques (§ 2) to use these codeword statistics as a proxy to understand the characteristics of the physical-layer transmission errors.

We make several observations that are relevant to network operations and research. First, we find that the average codeword error rate of an FN in our data spans six orders of magnitude, ranging from $1.3 \times 10^{-6}\%$ to 4.51%. The middle 80% of the FNs (excluding the top and bottom 10%) have average codeword error rates ranging from $9.53 \times 10^{-6}\%$ to 1.34×10^{-3} %. We establish a relation between the codeword error rates in our data and the packet loss rates from FCC's MBA data, by assuming that the cable ISPs included in the MBA study have similar physical layer characteristics. We find that, for the five cable ISPs MBA monitors, even with a conservative estimate, 12% to 25% of the packet losses could come from the physical layer.

This finding has several ramifications. First, it establishes a baseline for a "normal" physical-layer error rate. If an ISP's packet loss rate significantly exceeds the baseline, it either indicates that there is an anomaly in the network infrastructure, or the congestion loss is high. Second, it challenges the assumption of loss-based congestion control protocols, as a significant percentage of packet loss can be attributed to physical layer errors even in wireline networks. Lastly, it suggests that comprehensive network measurements should use

packets of different sizes to measure packet loss, as codeword error loss is not negligible and packets of different sizes would be encoded in different numbers of codewords, resulting in different loss rates.

A second noteworthy finding is that we observe in a small number of FNs, all devices in those networks show codeword error rates exceeding 1% for months of time. Surprisingly, customers served by these devices do not make more trouble calls on average. In contrast, when customers who reside in the networks with a typical codeword error rate experience an error rate of the same value ($> 3\%$), they make nearly 15 times more daily customer calls. This discovery suggests that codeword error rates can reliably detect network faults in the absence of customer trouble tickets and ISPs should not solely rely on customer tickets to detect network maintenance issues. Based on this discovery, the ISP we collaborate with has developed an internal tool to periodically monitor codeword errors across its networks. In addition, the observation that codeword error rates of $> 1\%$ may persist for months suggests that congestion may not be the culprit when users experience poor application performance.

Finally, we analyze how codeword error rates change before and after COVID-19 and find that the error rates are not impacted by the increase in traffic loads. We find that the codeword error rates of devices in the same FN are more correlated when their codeword error rates are high. We also study how weather impacts codeword error rates. The results show that extremely high ($> 95°F$) or low ($< 15°F$) temperatures increase codeword error rates, while the types of precipitation (e.g., freezing rain, snow) tend to cause outages than increase codeword error rates.

A limitation of this work is that our findings are based on data from one ISP. That being said, the ISP that provides us the data follows standard industry practices and uses standard Cable Modem Termination System (CMTS) equipment from dominant vendors. While different ISPs may choose CMTS modulation profiles to overcome specific radio frequency (RF) impairments at the cost of potentially reduced capacity, we are not aware of other reasons that will cause the overall physical layer loss characteristics of one ISP to differ from those of others. We release the code and part of the data used for this study.[3]

To the best of our knowledge, this work is the first large-scale and public study on the characteristics of physical-layer transmission errors of cable broadband networks. We make three main contributions. First, we characterize the physical-layer transmission errors of 394 HFC network segments and establish the relationship between physical-layer transmission errors and packet loss measured by FCC's MBA project. Second, we show that physical-layer transmission errors can indicate network faults in the absence of trouble tickets. Finally, we show that codeword errors are not impacted by

---

traffic loads or types of precipitation, but tend to increase in extremely cold or hot weather.

2 Methodology

In this section, we describe how we estimate the physical layer transmission errors and how we relate them to upper layer packet loss.

2.1 DOCSIS Codeword

The data items used in this study are the DOCSIS codeword statistics. Before we describe the data, we first describe what DOCSIS codewords are and how they impact upper layer packet loss. A codeword is a cable modem's basic transmission unit at the physical layer. The cable modems used in this study are DOCSIS 3.0 modems. DOCSIS 3.0 uses Forward Error Correction (FEC) to detect and correct errors at the physical layer. A codeword includes a data section and an FEC parity check section. In DOCSIS 3.0, each codeword is generated using a Reed Solomon (RS) encoder. The size of a codeword can vary from 18 bytes to 255 bytes, containing k data bytes and $2T$ parity check bytes. An RS codeword with $2T$ parity check bytes can correct up to T byte or $8T$ bit errors [13]. Both the data length k and the parity check length $2T$ of a codeword are vendor and configuration dependent. Typically, a CMTS vendor specifies a default setting of k and T for a long codeword and a short codeword. Cable operators can choose different settings, but the current industry practice is to use the default settings chosen by vendors.

Most of our data are collected from CMTS devices manufactured by a dominant vendor in the U.S. As an example, the default setting for our data includes two codeword lengths: one long codeword and one short codeword. The long codeword has a data length k of 200 bytes and a parity check byte length of $2T = 30$ bytes. The long codeword is able to correct 15 bytes of errors. Similarly, the short codeword has a data length $k = 99$ bytes, and a parity check byte length $2T = 10$ bytes. It is able to correct 5 bytes of errors in a codeword.

When a cable modem receives a data frame from an upper layer protocol such as Ethernet, if the data frame fits into a long or a short codeword, it will transmit the data frame using one codeword. Otherwise, the modem will use multiple codewords to transmit the data frame. A cable modem will at most use one short codeword at the end of a data frame to transmit it. If a data frame does not fit exactly into multiple codewords, the cable modem will use padding bytes at the last codeword. Figure 1 shows an example of how a cable modem encodes an Ethernet MAC frame into multiple codewords.

As bit errors at the physical layer tend to be bursty, a cable modem uses a scrambler to permute the content of a codeword before transmission, following a pre-defined pseudo-random pattern. The receiving end, the CMTS, will reverse the permutation before decoding the received data. Therefore, bursty errors in transmitted signals become random errors in the unscrambled codewords.
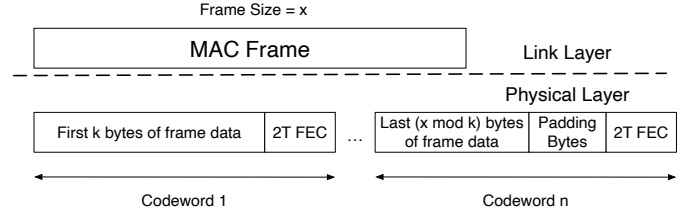
Figure 1: **A link layer MAC frame is encoded by multiple codewords at the physical layer. Each codeword has a data section and an FEC section.**

2.2 Codeword Error Rate

We refer to the ISP that provides us data as AnonISP. We now describe how we compute the codeword error rate using data collected by AnonISP. AnonISP collects the data through their Proactive Network Management (PNM) platform [11], which is part of DOCSIS's design. It aims to help cable ISPs troubleshoot and diagnose their networks. With PNM, an ISP can collect various performance metrics from both cable modems and a CMTS, including codeword counters, signal transmission power (TX power), and signal to noise ratio (SNR).

For each cable modem, AnonISP collects the total number of unerrored codewords sent by a cable modem, the number of uncorrectable codewords that fail FEC, and the number of codewords corrected by FEC periodically. All numbers are cumulative since the modem's last reboot. From DOCSIS 3.0's specification [13], uncorrectable codewords are discarded without link-layer retransmission. For correctable codeword errors, they do not manifest them as upper-layer packet discards. Therefore, we focus on the uncorrectable codeword errors. Without specific clarification, in this work, we use *codeword errors* to refer to uncorrectable codeword errors.

We estimate the average codeword error rate $P(e)$, where e denotes packet loss events, of a cable modem as the number of *uncorrectable* codewords divided by the *total* number of codewords a CMTS receives in each collection period: *uncorrectable/total*. Note that in our data, codewords are of two different lengths. The codeword error rate of a long codeword and a short codeword could be different, but we can still estimate the average codeword error rate without knowing the distribution of short or long codewords. Formally, let $P(e|l)$ denote the probability of a long codeword error rate and $P(e|s)$ be the probability of a short codeword error rate. Let $P(l)$ and $P(s)$ be the probability of a long or short codeword occurring in the data stream, respectively. We can compute the average codeword error rate $P(e)$ as follows:

$$
\begin{aligned}
P(e) &= P(e|l)P(l) + P(e|s)P(s) \\
&= \frac{uncorrectable\ long}{long}P(l) + \frac{uncorrectable\ short}{short}P(s) \\
&= \frac{uncorrectable\ long}{long} \times \frac{long}{total} + \frac{uncorrectable\ short}{short} \times \frac{short}{total} \\
&= \frac{uncorrectable}{total}
\end{aligned}
$$

2.3 Codeword Errors vs. Packet Loss Rates

We aim to understand how physical layer transmission errors affect end-to-end packet loss. We ask this question: *how much do physical layer transmission errors contribute to higher-layer packet loss*? Since FCC's MBA project measures packet loss on broadband networks, if we can establish the relationship between codeword errors at the physical layer and packet loss measured by the MBA project, then we can estimate how much packet loss is caused by physical layer errors. To do so, we make the assumption that the physical layer loss characteristics of AnonISP's networks are representative of those of U.S. cable broadband networks. With this assumption, we can correlate the network-layer packet loss rates from the MBA project with codeword error rates observed in our data.

There are three challenges in establishing the correlation. First, a packet has a variable length so that it may be encoded in multiple codewords. Hence, there does not exist a one-to-one correspondence between the codeword error rate and the packet loss rate. Fortunately, FCC's MBA project uses short UDP ping packets with packet length set to 62 bytes [4] to continuously monitor the packet loss rates. Such a packet will be encoded using one short codeword under common CMTS configurations. Therefore, if we assume cable broadband networks operate in similar physical conditions, then the short codeword error rate will correspond to the packet loss caused by physical layer errors in the MBA project.

Second, the codeword error rate we measure is the average codeword discard rate that includes both short and long codewords, while the FCC MBA project uses only short UDP packets that correspond to short codewords for measuring packet loss. To address this challenge, we analyze whether the average codeword error rate is an over- or under-estimate of the short codeword error rate. According to the common CMTS configurations, for a long codeword to become uncorrectable, more than 120 bits out of 200 bytes must be errored. For a short codeword to become uncorrectable, more than 40 bits out of 99 bytes must be corrupted. Since a long codeword is roughly twice the size of a short codeword, and the number of FEC bits in a long codeword is three times that in a short codeword, the long codeword should have a much lower error rate than the short one, assuming the bit error rate in a long or a short codeword is the same. Therefore, the average codeword error rate in our data is a lower bound to the short codeword error rate. In other words, if the cable networks operate in similar physical conditions, the average codeword error rate we measure is a lower bound to the physical layer error rate from FCC's MBA project, since the measurement project only uses short UDP packets.

Third, the codeword statistics we obtain only include the upstream channels. That is, we only observe the codeword errors from a customer's device to the ISP's cable headend. However, packet loss measured by FCC's MBA project is bi-directional. To reconcile the difference, we use the upstream transmission errors as a lower bound to bi-directional transmission errors and an upper bound to downstream transmission errors. DOCSIS's downstream channels operate at higher frequencies than upstream channels [13], while RF noises concentrate on the lower RF range. In § 4.1, we show how the codeword error rate decreases as a channel's frequency increases. Therefore, downstream channels should have lower codeword error rates than upstream channels and we can use the upstream transmission errors to upper-bound downstream transmission errors.

3 Datasets

Next, we describe our datasets and data cleansing steps.

AnonISP Data   At the time the data were collected, AnonISP uses three upstream channels and sixteen downstream channels as data channels in their networks. Each channel is of 6MHz width. Our data include the upstream codeword statistics only. At each data collection time point, AnonISP collects several metrics for each upstream channel, including SNR, cumulative values of the number of unerrored codewords, the number of corrected codewords, and the number of uncorrectable codewords each cable modem sends to a CMTS since it reboots, and the signal transmission (TX) power of a cable modem. The data is collected from 01/06/2019 to 03/03/2020 and from 03/24/2020 to 04/17/2020. The data are collected every 4 hours. In total, the data come from 77,696 devices and span 394 fiber optical nodes. On average, each fiber node has 197 devices. In total, we have collected ~ 139M data points for each upstream channel. We call this dataset the codeword dataset.

In the codeword dataset, each data point contains the data from three upstream channels. If these three channels send fewer than 2,000 codewords in total between the current and its previous data collection point, which means the three channels send less than 200KB of data during the last 4 hours, we will consider the current data point invalid as the loss statistics may be distorted because of too few numbers of codewords.

In addition, it is possible that at a data collection point, we fail to retrieve data from a device. Since our data are collected every 4 hours, if we observe that the time interval between two adjacent data points is $4 \times (1 + x)$ hours (where x rounds to an integer), we will insert x empty placeholder data points in the data stream. These empty placeholders indicate that we fail to retrieve data at those time points. We infer empty placeholder data points and refer to them as *missing data*. If all three channels' data are missing, we will count this data point as a *missing* data point, which can indicate that the network is unavailable. If only one or two channels have missing data, we will discard this data point, because we often combine the three channel's data for our analysis.

Among all of the data points (~ 139M), we discard ~ 33M data points and obtain ~ 106M valid data points. In addition, we infer ~ 11M missing data points when a collection point fails to collect any data. Among the 33M discarded data

points, ∼ 9M of the data points are discarded due to missing partial channel data, while ∼ 24M of the discarded data points have fewer than 2,000 codewords. We use the valid data points and the missing data points for our analysis in this paper.

Besides the codeword dataset, AnonISP also provides us the customer call trouble tickets from the same group of devices during the same time periods. Each trouble ticket includes the call time, the description of the issue that triggered the customer call, and how AnonISP resolved the issue.

FCC Data from MBA Project To understand the relation between codeword error rates and packet loss rates, we compare our data with the FCC data obtained from the MBA project [2]. The FCC data are continuously collected from thousands of users all over the United States since Jan 2011 and are available to the public. FCC deployed whitebox measurement devices in volunteers' homes. The volunteers are distributed across 10 wireline broadband providers. The measurement devices continuously send UDP packets to target test nodes to measure packet loss rates. If a device does not receive a response packet within three seconds, it labels the packet as lost. The devices follow the Poisson distribution to send probe packets over a fixed interval of one hour [4]. We use the FCC data collected from the same period as our data. The FCC data contain several broadband technologies, including DSL, Cable, and FTTH. Since our data are from cable networks only, we only analyze the data collected from cable networks in the FCC data, and leave a comparison among different physical layer techniques as future work.

Weather Data We collect weather data that overlaps with the codeword dataset in time and location to study how weather affects physical layer transmission errors. We use the IBM Weather Data APIs [6] to collect the hourly weather conditions given a time period and the zip code each device in our dataset belongs to. Each weather data record includes the basic weather metrics such as temperature, atmospheric pressure, and humidity. It also contains the description of the current weather type, such as Light Rain or Snow.

Ethical Considerations Prior to obtaining data from AnonISP, we consulted with our organization's IRB and obtained their permission to conduct this research. The MAC address and account number provided by AnonISP are hashed values. All the statistics in our data are performance monitoring metrics generated by the devices. For each customer trouble ticket, it only records the time of the trouble call, the hashed account number to match the monitoring metrics, the description of the issue, and the action of the ISP. So there is no personal identification information included in our data.

4 Physical Layer Loss vs. Overall Loss

In this section, we study how the physical layer errors look like using the codeword dataset. We compare the physical layer errors with packet loss observed in the FCC data, aiming
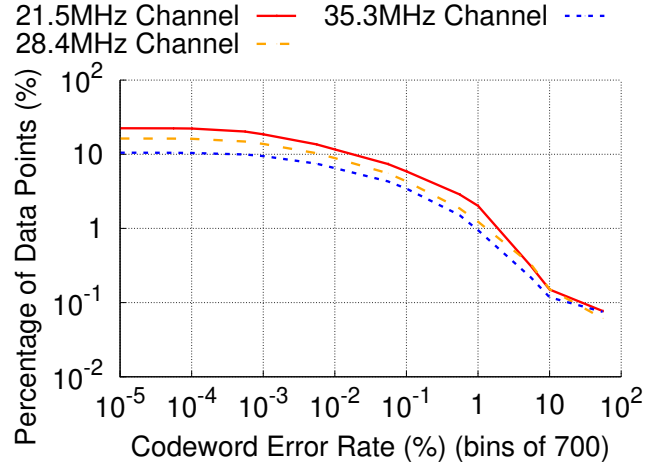
Figure 2: **The complementary cumulative distribution of the codeword error rate in each upstream channel. The error rate decreases when the channel frequency increases.**

to answer the question: *What is the relationship between the physical layer codeword error rate and the end-to-end packet loss rate?*

4.1 Codeword Errors in Different Channels

Our codeword data are collected from three upstream channels in AnonISP's HFC networks. The three upstream channels send RF signals with 21.5MHz, 28.4MHz, and 35.3MHz center frequency, respectively. The downstream channels will each use a higher center frequency, ranging from 54 MHz to as high as 1000 MHz. Figure 2 plots the complementary cumulative distributions of all three channels' codeword error rates, respectively. Each data point is computed as the number of uncorrectable codewords divided by the total number of codewords a device sends to a CMTS over the 4-hour data collection period. Both the x-axis and y-axis are in log-scale.

From Figure 2, we can see that the majority of the data points have no or few codeword errors, as seen in the flat sections at the beginning of the lines. At least 75% of the data points in each channel have no uncorrectable codewords. However, for all three channels, the curves start to drop after the error rate exceeds 10^{-4}, suggesting that a small fraction of lossy periods contribute to the majority of codeword errors. In particular, more than 1% of the data points have codeword error rates exceeding 1%; and more than 0.1% of the data points have codeword error rates exceeding 10%. The average codeword error rates in the three channels are 0.11%, 0.08%, and 0.06%, respectively. Furthermore, the channels with a higher center frequency have lower error rates, consistent with the operational knowledge that lower frequency channels are more prone to RF interference.

In the DOCSIS design, a cable modem will switch to a different upstream channel when one upstream channel does not work expectedly. We study how codeword errors in the upstream channels are correlated. That is, for each data point, we investigate whether the three channels show similar codeword
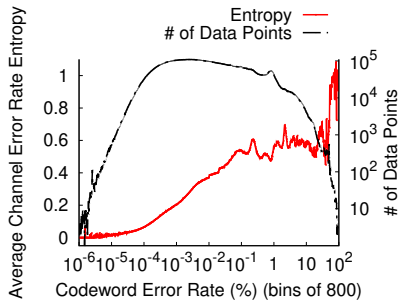
Figure 3: **The relationship between the average channel error rate entropy and the codeword error rate together with the number of data points in each bin.**
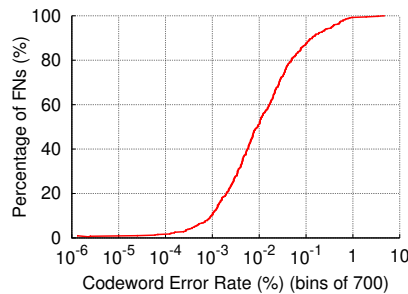
Figure 4: **CDF of the average codeword error rate of an FN.**
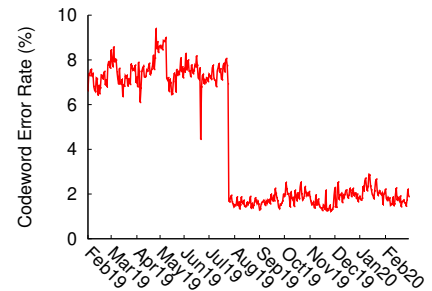
Figure 5: **The daily-average codeword error rate of the FN with the highest average codeword error rate in our data.**

error rates or the codeword error rates of the three channels differ by a lot. To quantify the similarity of codeword errors in each data point, we compute the channel error entropy S, where S is defined as the error rate entropy of each data point. That is, let s_i be the uncorrectable codewords sent via channel i divided by the total number of uncorrectable codewords across all three channels in each data point. We compute $-\sum_{i=1}^{3} s_i \ln s_i$ for each data point. A higher channel error entropy value indicates a modem has a lower error rate variation among the three channels for this data point. If all the uncorrectable codewords are from one single channel, then S will be 0. In contrast, if the three channels have the same codeword errors, the value of S will achieve its maximum.

Figure 3 shows the relationship between the average channel error rate entropy and the codeword error rate. We divide the codeword error rate values into 800 bins and calculate the average error rate entropy of data points in each bin. This figure shows that the average channel error rate entropy increases as the codeword error rate increases, suggesting that when codeword errors in one upstream channel occur, they are likely to occur in other upstream channels as well. So DOCSIS's upstream channel switching algorithm may be insufficient to avoid codeword errors.

Figure 3 shows that codeword errors are highly correlated in three upstream channels when the codeword error rate exceeds 0.1%. Therefore, without specific mentioning, we will use the number of combined codewords and the number of combined codeword errors from three channels in each data point for our analysis in the rest of this paper. The average codeword error rate from the combined channels is 0.088%, while 98.68% data points have a codeword error rate < 1%.

Takeaways: Codeword errors occur infrequently, and a small percentage of lossy periods contribute to most of the codeword errors. Higher frequency channels have lower codeword error rates, and when codeword errors occur, they tend to occur in all upstream channels. We show more examples of raw codeword error rates in Appendix A.

Next, we investigate *whether the devices in different fiber*

optic nodes (FNs) will have different codeword error rates. To do so, we compute the codeword error rate of each device over the 16-month data collection period. We then compute the average codeword error rate of each FN by averaging the codeword error rates of all devices in the FN.

Figure 4 shows the CDF of the average codeword error rate among 394 FNs in our data. The x-axis is again in log-scale. We observe that the average codeword error rate differs significantly among different FNs. In our data, there are three FNs that have an average codeword error rate higher than 1%, and 46 FNs have an average codeword error rate between 0.1% to 1%. We define the FNs with > 1% error rates as unhealthy FNs, the FNs with 0.1% - 1% error rates as alarming FNs, and the remaining 345 FNs as healthy FNs. The thresholds 1% and 0.1% are chosen according to operational experience suggested by AnonISP. The healthy FNs constitute 87.6% of the FNs seen in our data and the codeword error rate averaged over those FNs is 0.0179%. They contribute to only 18.79% of the total codeword errors. In contrast, the alarming FNs are 11.68% of the FNs see in our data and their average codeword error rate is 0.352% and they contribute to 42.67% of the total codeword errors. The unhealthy FNs are 0.761% of the total FNs. Their average codeword error rate is 3.778% and they contribute to 38.54% of the total codeword errors.

We are interested in understanding why certain FNs have such high codeword error rates. Figure 5 shows the daily codeword error rate from the FN with the highest average codeword error rate in our data. The average error rate is over 7% before July 2019, and then it decreases to 2%, suggesting that AnonISP repaired some problems in this node. However, even after this repair event, this FN still has a nearly 2% daily codeword error rate, and it exists till the end of our data. We are informed by AnonISP that this FN is affected by issues in the network hardware. Its maintenance team is aware of this persistent problem, but it either cannot repair the issue for some reason (e.g., waiting on permits, access and/or restrictions, etc.) or has deprioritized the repair for some reason (e.g., it is a known but uncorrectable cause).
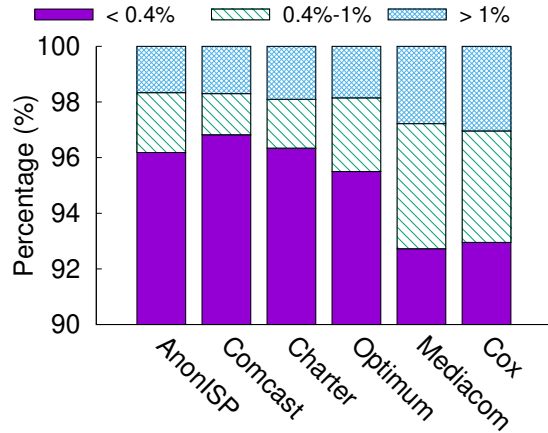
Figure 6: **This figure shows the percentage of data points whose packet loss rate was less than 0.4%, between 0.4% to 1%, and greater than 1% for each cable ISP in the FCC data, together with AnonISP's loss rate we measure.**
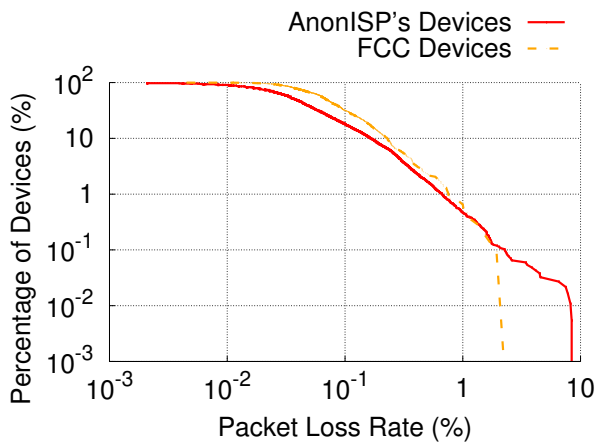
Figure 7: **This figure shows the complementary cumulative distribution of the average packet loss rate of each device in the FCC data and in the AnonISP's data we measure, respectively.**

Takeaways: Codeword error rates in different HFC network segments vary significantly. Some network segments may experience persistent high codeword error rates ($> 1\%$). There are 87.6% healthy FNs in our data and they contribute to 18.79% of the total codeword errors. The 12.4% alarming and unhealthy FNs contribute to 81.21% of codeword errors seen in our data.

4.2 Comparison to FCC data

A key question this work aims to answer is how much physical layer error loss contributes to end-to-end packet loss. We compare our data with the FCC data collected by the MBA project to gain insight into this question. The FCC data measures the packet loss rates in different types of networks, including Cable, FTTH, and DSL. We only used the data collected from cable ISPs in the FCC dataset.

The FCC dataset does not include data from AnonISP, which prevents us from comparing the physical layer loss of
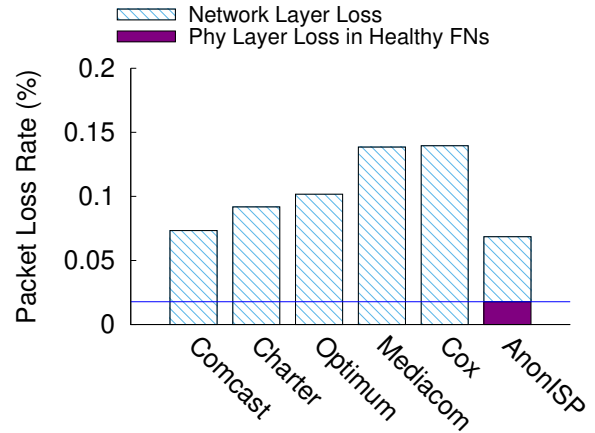
Figure 8: **The average packet loss rate in each cable ISP, together with the packet loss rate we measure in AnonISP and the physical layer error rate among the health FNs in AnonISP.**

AnonISP's networks directly with end-to-end packet loss. To address this challenge, we design an experiment to approximate the FCC's packet loss measurement for AnonISP. We deploy a measurement node on a vantage point that is close in router hops to AnonISP's networks. The vantage point sends ICMP echo request packets to all 18,772 pingable modems located in AnonISP's cable networks in the data collection regions periodically. For each modem, we send ~ 250 ICMP packets per hour. We run this experiment from 11/03/2021 to 11/11/2021. The measurement node sends 914M packets in total. The FCC dataset has been cleansed to exclude data points with high loss rates ($> 10\%$) and high RTTs [5]. We applied to our measurement results the same data cleansing script as applied to the FCC data.

Figure 6 shows the percentages of end-to-end packet loss rates in different ranges for the FCC dataset, together with the end-to-end packet loss rate we measure for AnonISP. We note that the ICMP packet loss rate observed in AnonISP is comparable to the packet loss rates observed in the FCC measurement. Similar to our measurement results, most data points in the FCC data suffer no or few packet losses and the majority of packet loss comes from a small percentage of lossy periods. For example, around 97.03% data points in our data have less than 0.4% loss rate, and 1.73% data points have loss rates between [0.4%, 1%], while in Comcast's data, 96.43% data points have less than 0.4% packet loss rate, and 1.72% data points have packet loss rates between [0.4%, 1%].

We compute the average packet loss rate for each device in the FCC data and that for each device in our measurement. There are a total of 1,073 devices installed in five cable ISPs in the FCC measurement. In contrast, we measure 18K+ devices. Figure 7 shows the results. Since we measure more devices, we see a wider range of packet loss rates in our measurement than that in the FCC data.

Figure 8 shows the average packet loss rate in each cable ISP, together with the average physical layer error rate

among the healthy FNs seen in our data. We compare the FCC packet loss rate with the physical layer codeword error rate from healthy FNs only because FCC's volunteers are sparsely located. The alarming and unhealthy FNs account for only 12.44% of the 394 FNs in our data. Therefore, there may or may not be any devices located in those outlier FNs in the FCC study. As we aim for a lower-bound estimate regarding the physical layer's contribution to end-to-end packet loss, we exclude the FNs with the high error rates in our data from the comparison. We assume that the physical layer error rates in those ISPs' healthy FNs are the same as those in our data (0.0179%) since we think our data is representative of the nature of cable networks. Based on this assumption, we see that at least from 12% to 25% packet loss seen in the FCC data could have come from physical layer errors.

Meanwhile, we estimate how much ICMP packet loss from our own measurement could come from physical layer transmission errors. If we assume that our sampled devices do not include any devices in the alarming or unhealthy FNs, then 26.1% of the packet loss seen in AnonISP can be attributed to physical layer codeword errors. However, this estimate may be overly conservative, as we receive ICMP echo replies from more than 20% of all devices in our codeword dataset. If we assume that we have representatively sampled devices from both healthy and alarming FNs, and since the average codeword error rate among the healthy and alarming FNs in our data is 0.0551%, and the average ICMP packet loss rate we measure is 0.0686%, then 80.3% of packet loss in our measurement could have come from physical layer transmission errors. However, since we cannot establish a one-to-one correspondence between a device we ping and a device we see in the codeword dataset, due to the anonymization procedure applied to the data, we cannot conclude whether the devices we ping are a representative subset of devices from the healthy and alarming FNs. Therefore, we prefer to use 26.1% as a safer lower bound.

Our estimate presents how much physical layer errors contribute to end-to-end packet loss. This is a lower bound estimate for the following reasons. First, the baseline codeword error rate we compute is the average between long and short codewords, while the FCC measurement packets only use short codewords (§ 2.3). The short codeword's error rate is higher than the average codeword error rate due to the encoding scheme. Second, the baseline codeword error rate only includes codeword errors in the upstream channels, while the packet loss measurement is affected by both upstream and downstream errors. Third, we have conservatively excluded all alarming and unhealthy FNs in our codeword error rate calculation, while the FCC measurement may include such nodes. Finally, the baseline codeword error rate only includes the network segment from a cable modem to a CMTS, while the physical layer errors in the entire end-to-end paths can contribute to packet loss in the FCC measurement.

We also note that different cable ISPs have very different
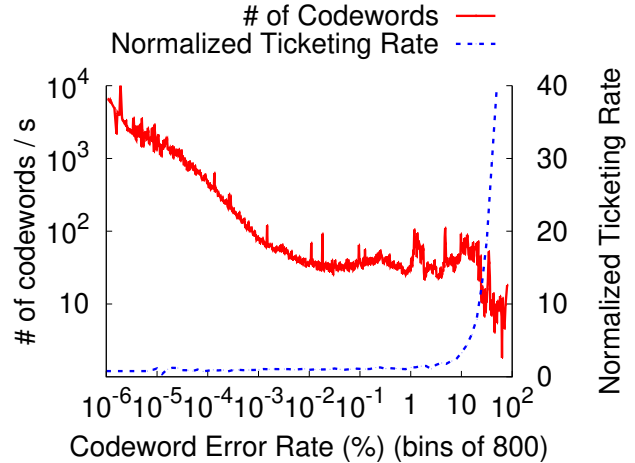
Figure 9: **This figure shows how codeword error rate affects the number of codeword sent per second by a device and the normalized customer ticketing rate. The number of data points in each bin is the same as shown in Figure 3.**

packet loss rates. Among the five ISPs in the FCC data, Comcast shows the lowest average packet loss rate: 0.073% with a standard deviation of 0.486%. In contrast, the average packet loss rates of Mediacom and Cox are 0.138% (with a standard deviation of 0.621%) and 0.140% (with a standard deviation of 0.619%), respectively. They are almost two times higher than Comcast's average packet loss rate. AnonISP shows the lowest packet loss rate among the six ISPs. Its packet loss rate is slightly lower than Comcast's. We speculate that this is because we place the measurement node close to the cable modems and our measurement packets have shorter RTTs than FCC's measurement packets. Therefore, they encounter fewer congestion and physical layer transmission error events.

Takeaways: We show that 12% to 25% of the packet loss measured by FCC's MBA project on cable ISPs could have come from physical layer errors. This result suggests that physical layer errors in cable networks play a non-negligible role in end-to-end QoS. Network research and operations should take this source of packet loss into account.

5 Analysis of User Behavior

In this section, we investigate how codeword errors affect user behavior. We use the amount of data sent by customer devices and the customer reported trouble tickets to quantify user behavior and study how they change when codeword error rates change.

5.1 Impact on Usage

We use the number of codewords in each data point to estimate the amount of data users sent, because application data will be sent using codewords. We divide the range of codeword error rates into 800 bins. For each bin, we calculate the total number of codewords from the data points falling into this bin and normalize it by the period of time covered by the data points in the bin. With this computation, we obtain the
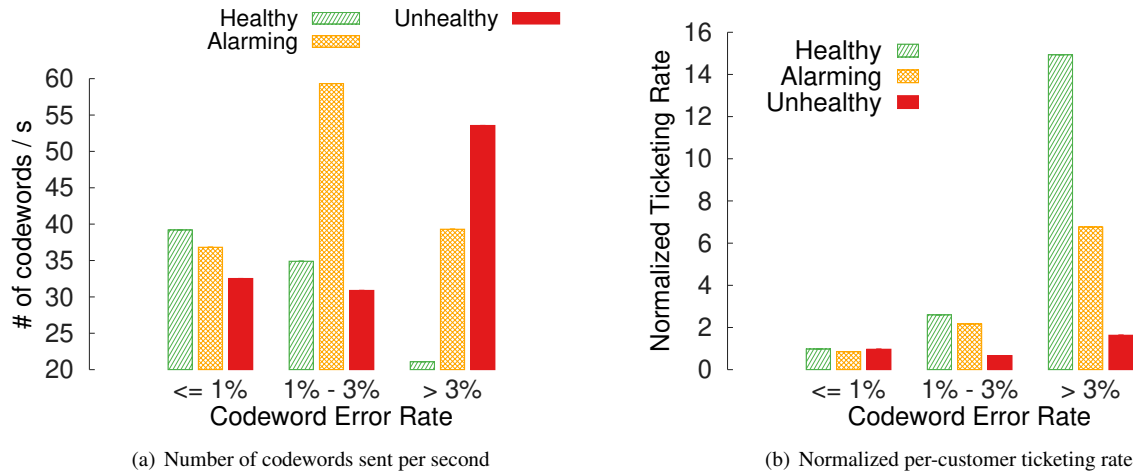
(a) Number of codewords sent per second

(b) Normalized per-customer ticketing rate

Figure 10: **This figure shows how users or devices in different types of FNs behave when experiencing different codeword error rates.**

number of codewords sent by a device per second when a specific codeword error rate occurs.

Figure 9 shows the relationship between the codeword error rate and the number of codewords sent per second by a customer device. We can see that the number of codewords sent per second by a user device decreases as the codeword error rate increases from $10^{-6}\%$ to $10^{-2}\%$. It plateaus between $10^{-2}\%$ and 1%, and sharply decreases when the codeword error rate increases beyond 10%. The data between 1% and 10% error rates are jagged. One plausible explanation is that loss rates within this range will significantly impact user experience [2, 3], and users or applications may react to the adverse conditions by multiple retries, leading to a fluctuated data rate.

5.2 Impact on Customer Trouble Tickets

When a customer has poor QoE, she may call her ISP's customer service to report the issue. Therefore, customer tickets are a good indicator of network problems and also reflect customer experience [22]. We study how customer reported trouble tickets are affected by codeword errors. We define the *ticketing rate* as the average number of trouble tickets each customer reports in a unit time. We compute a baseline ticketing rate by computing the average number of tickets reported by each customer in a unit time. We define a *normalized ticketing rate* as a ticketing rate divided by the baseline ticketing rate.

Figure 9 shows the relationship between the normalized ticketing rate and the codeword error rate. Similarly, we divide the range of codeword error rates into different bins and compute the normalized ticketing rates for data points that fall into each bin. The customer ticketing rates remain stable until the codeword error rate exceeds 1%. It increases fastly after that. In extreme cases, when the codeword error rate exceeds 50%, the customer ticketing rates increase by more than 40

times compared to the baseline ticketing rate.

5.3 Conditioned User Behavior

Next, we investigate how codeword error rates impact a user's behavior for users located in different network environments. From our study in § 4.1, we show that different FNs can have drastically different codeword error rates. We classify the FNs in our data into three types based on their average codeword error rates: healthy ($< 0.1\%$), alarming ($[0.1\%, 1\%]$), and unhealthy ($> 1\%$). We divide the codeword error rates into three ranges $< 1\%$, $[1\%, 3\%]$, and $> 3\%$ and examine how user behavior in different types of FNs varies in different codeword error ranges. Specifically, we compute the number of codewords sent per device and the normalized ticketing rate for each codeword error range for healthy, alarming, and unhealthy FNs, respectively.

Figure 10(a) and 10(b) show the results. For users in healthy FNs, their data usage decreases and their normalized ticketing rate increases as the codeword error rate increases. This trend is consistent with the general trends shown in Figure 9.

The usage and ticketing rate patterns in the alarming and unhealthy FNs are somewhat counter-intuitive. Customers in unhealthy FNs report much fewer tickets on average, when their networks show a $> 3\%$ loss rate. In contrast, for the customers in healthy FNs, when the codeword error rate is larger than 3%, the probability of a customer reporting a ticket will increase by 14.93 times. Instead, the customers in unhealthy FNs increase their data usage when the error rate exceeds $> 3\%$, suggesting that they or their applications attempt to use retransmissions or redundant transmissions to overcome packet loss. For customers in alarming FNs, their behavior is even more puzzling. They increase their data usage when the error rate is in the $[1\%, 3\%]$ range and decrease the usage when it exceeds 3%. One plausible explanation is that the customers in those FNs would attempt retry or retransmission
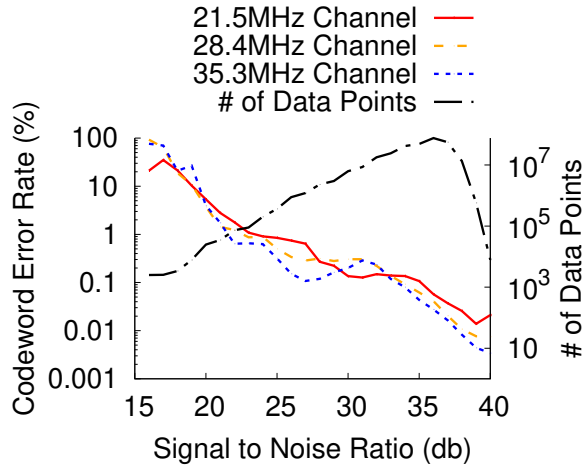
Figure 11: **The correlation between the codeword error rate and the SNR together with the number of data points with respect to an SNR value.**
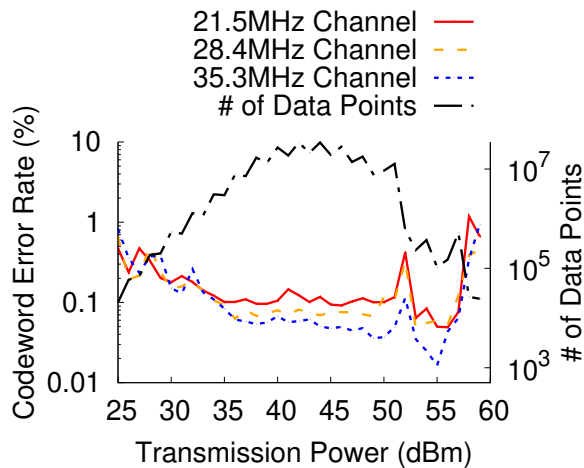
Figure 12: **The correlation between the codeword error rate and the TX power together with the number of data points with respect to a TX Power value.**

first when the network conditions slightly worsen, but will give up using the networks when the network conditions are significantly worse than what they are used to.

Takeaways: Users generally report more trouble tickets when the codeword error rate increases. However, users belonging to an FN with a consistently high codeword error rate have a higher tolerance for packet loss. This result indicates that network operators should continuously monitor the codeword error rates of their networks. Lack of trouble tickets alone is not a reliable indicator of good network conditions.

6 What Affects Codeword Error Rate?

In this section, we study what factors impact codeword errors. We examine how other PNM metrics (SNR and TX power) correlate with codeword error rate, how the traffic load increases after COVID-19 and different weather conditions affect the codeword errors in an HFC network, and how
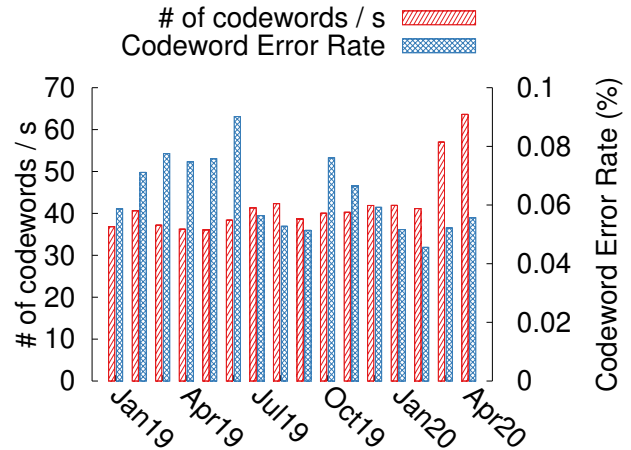
Figure 13: **This figure shows the average number of codewords sent per second and the average codeword error rate in each month from Jan 2019 to Apr 2020.**

the codeword errors of different devices correlate with each other.

6.1 SNR and TX Power

Figure 11 shows the correlation between the SNR and the codeword error rates. For each data point, we plot the SNR value on the x-axis and the y-axis is the codeword error rate of each upstream channel in log-scale. The average codeword error rate decreases as the SNR increases, indicating that codeword errors are caused by noises breaching into a cable segment. Figure 12 shows the correlation between the transmission (TX) power and the codeword error rate. The average codeword error rate shows a decreasing trend as the TX power increases until 52 dBm. However, the error rates peak when the TX power reaches 52 dBm or 58 dBm, respectively. This is because the modems have reached their maximum TX power. Different modems have different maximum TX power settings. Some of them have their maximum TX Power set to 52 dBm, while some modems have it set to 58 dBm or higher. Figure 12 shows that a modem increases its TX power in response to codeword error rates and codeword error rates will spike when a modem cannot outpower the noises in a cable segment.

6.2 Traffic Load

Our codeword dataset includes data collected from January 2019 to April 2020. During the last two months of the data collection period, COVID-19 hit U.S. and remote learning/working started. We break the data points into different months to plot the monthly average number of codewords sent by each device per second and the monthly average codeword error rate. Figure 13 shows the results. We observe that the number of codewords sent per second increases by 44.36% and 61.11% in March 2020 and April 2020, respectively. In contrast, the monthly codeword error rate fluctuates over time, and we do not see a significant increase or decrease in March
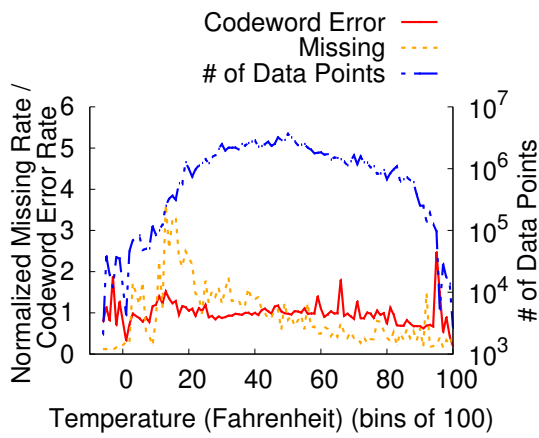
(a) Temperature
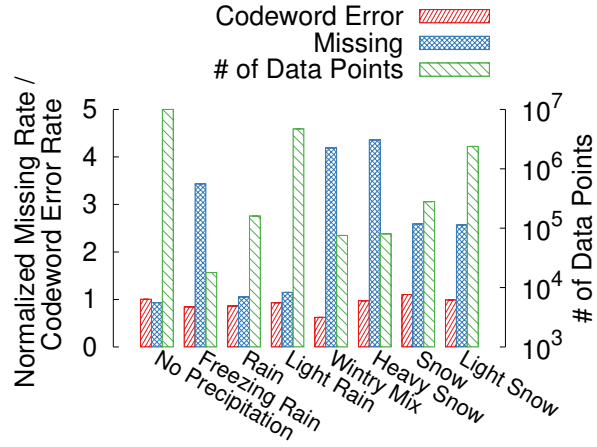
(b) Precipitation Type

Figure 14: **This figure shows how codeword error rates and network (un)availability (captured by the normalized data missing rate) are affected by temperature and type of precipitation.**
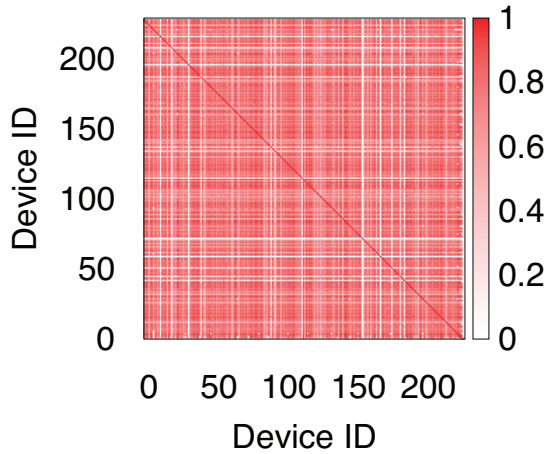
Figure 15: **The correlation matrix in the FN with the highest average codeword error rate. Each Device ID represents a modem. This figure shows the codeword error rates of modems in the FN with the highest average codeword error rate are highly correlated.**

2020 and April 2020.

6.3 Weather

Padmanabhan et al. [24] have shown that severe weather conditions reduce the availability of residential networks. We are interested in finding out whether severe weather will impact the codeword error rate, which is a network reliability metric. As described in § 3, we collect the historical weather data that overlap in time and location with our codeword data. We compute the codeword error rates under different weather conditions. For comparison, we use the data points where no performance data are collected as indicators of networking being unavailable. We compute the rate when this event happens and refer to it as the missing data rate.

Figure 14(a) shows how the codeword error rate and the

data missing rate change as the temperatures change. For clarity, we normalize the codeword error rate with the average codeword error rate among all FNs seen in our data . Similarly, we normalize the data missing rate. In Figure 14(a), we see the codeword error rate increases when the temperature is around $10°F$, $< 0°F$, or just below $100°F$. We do not have many data points for $< 10°F$ or $> 100°F$ weather. So the data points in those regions may not be representative. We see that the data missing rate increases significantly when the temperature is between $10°F$ and $30°F$, consistent with the results in [24].

Figure 14(b) shows the normalized codeword error rate and normalized missing rate in different weather types. The data missing rates are significantly higher in Freezing Rain, Wintry Mix, and Heavy Snow weather types. One explanation
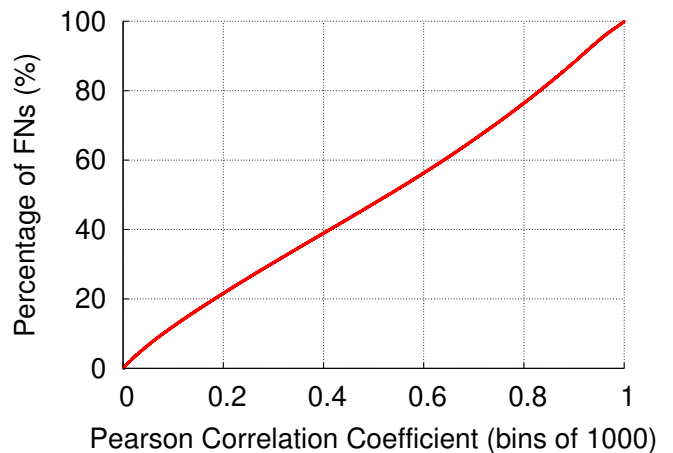
Figure 16: **This figure shows the CDF of the average Pearson correlation coefficient of each FN, which is averaged over all devices' pair-wise Pearson correlation coefficients. In about 30% of FNs, the devices show strong error rate correlation (> 0.7).**
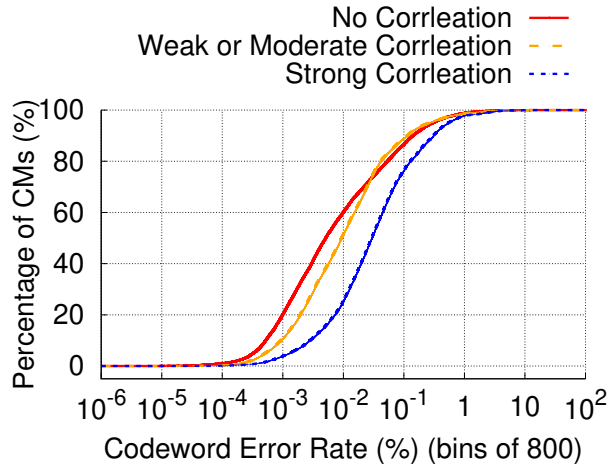
Figure 17: **This figure shows the CDF of the average codeword error rates of devices in different correlation groups. The devices that show strong error rate correlation with other devices in the same FN tend to have higher codeword error rates.**

we learn from AnonISP is that some HFC networks use aerial cables and these weather types can cause damage to those cables. The weather type's impact on the codeword error rate is not as clear. The three weather types that significantly increase network unavailability: Freezing Rain, Wintry Mix, and Heavy Snow do not significantly increase the codeword error rate. This indicates the codeword error rate may not be affected by the types of precipitation.

6.4 Codeword Error Correlation

Lastly, we analyze our data to answer this question: *How does a device's codeword error rate correlate to those of other devices in the same FN*? An HFC network is a shared medium network. Devices connected to the same FN may share RF impairments. Understanding the scope of RF impairment sharing can help us develop future fault diagnosis tools.

We quantify the correlation of codeword error rates of two devices using the pair-wise Pearson correlation coefficient [9]. For each device, we compute its codeword error rate at each data collection point and treat it as an element in the input vector to the Pearson coefficient calculation. Since our data span a 16-month period and each data point is collected every four hours, the length of the vector is around 2.5K. For each FN, we compute the pair-wise Pearson correlation coefficients for all devices in the FN. We then average the Pearson correlation coefficients among all devices in an FN to obtain the average Pearson correlation coefficient of the FN.

Figure 15 shows the correlation coefficient matrix for the FN with the highest average codeword error rate. The average correlation coefficient in this FN is 0.6798. According to [21], a correlation coefficient less than 0.3 indicates no correlation, between 0.3 to 0.7 means weak or moderate correlation, and larger than 0.7 shows a strong correlation. This figure shows that most devices in this FN have a strong error rate correlation. There are also some devices that do not have

any correlation with other devices, suggesting that these devices do not share the same RF impairments with the other devices.

Figure 16 shows the CDF of the average Pearson correlation coefficient in each FN. We observe that the devices in nearly 30% of the FNs show a strong correlation, indicating that for most of the time in these FNs, a large group of devices in the same FN share RF impairments. We also observe that nearly 30% of the FNs present no correlation among the devices.

We are interested in understanding how a device's codeword error rate is distributed when it shows a certain degree of codeword error rate correlation with other devices in the same FN. We divide the devices into three groups, No Correlation (average Pearson correlation coefficient < 0.3), Weak or Moderate Correlation ($0.3 \leq$ Pearson correlation coefficient < 0.7), and Strong Correlation (Pearson correlation coefficient ≥ 0.7). A device's average Pearson correlation coefficient is the sum of its pair-wise Pearson correlation coefficients with other devices in the same FN divided by the number of pairs.

Figure 17 shows the CDF of the codeword error rates for devices in different correlation groups. The x-axis is in log-scale. We see that for devices in the Strong Correlation group, the 10th and 90th percentiles of their codeword error rate distributions are $[2.82 \times 10^{-3}\%, 0.306\%]$; for devices in the Weak or Moderate Correlation group, the 10th and 90th percentiles of their codeword error rate distributions are $[9.32 \times 10^{-4}\%, 0.113\%]$; and for devices in the No Correlation group, the 10th and 90th percentiles of their codeword error rate distribution are $[5.21 \times 10^{-4}\%, 0.140\%]$. These results show that devices that show a high correlation to other devices are more likely to have high codeword error rates, suggesting that they are affected by the same RF impairments.

It is possible that an RF impairment only affects a portion of the devices in the same FN, which makes the average Pearson coefficient for the affected devices low since they have no correlation with the unaffected devices. In our future work, we plan to investigate whether a clustering algorithm based on codeword error correlation can help identify the devices that are affected by the same RF impairment.

7 Implications

We believe this work provides several implications for network operations and network research:

- When measuring packet loss on the Internet, one should vary the length of a measurement packet to gain a full spectrum of packet loss statistics. Packets of different sizes may be encoded into different numbers of codewords and experience different loss rates.
- When designing the network applications and protocols, one should take into account packet loss caused by physical-layer transmission errors in the RF systems. Exceptional

innovation in the cable broadband industry has allowed ISPs to use HFC networks to deliver high-speed data. But due to the RF range they operate in, transmission errors in those networks are not negligible.

- ISPs should not rely on customer tickets alone for network maintenance. Customers in chronically high-error-rate networks may have adapted to the network conditions.

8 Related Work

Last-mile Packet Loss: FCC launched the MBA project [2] in 2011 and has been publishing an annual report on broadband performance. FCC MBA project uses UDP pings to measure packet loss. We analyze the FCC data and estimate what fraction of packet loss from the FCC measurements is due to physical-layer transmission errors. Using the FCC data, Sundaresan et al. [28] show that different ISPs and different home network devices can lead to different latency and loss rate distributions. Sundaresan et al. [30] also show that for broadband network customers, the last-mile latency is the main bottleneck when visiting web pages since it significantly contributes to both DNS lookup time and the time to the first byte. Genin and Splett [18] use the download speed distribution from the FCC data to investigate where congestion happens, concluding that most of the Internet congestion occurs in the last-mile network.

In addition to the FCC MBA project, many researchers have also measured and characterized the reliability of the last-mile broadband access networks using their measurement apparatuses. Dischinger et al. [16] measure 1,894 broadband hosts from 11 ISPs with TCP and ICMP measurement packets. They show that both DSL and cable broadband networks exhibit non-negligible packet loss rates, with around 5% data points showing a loss rate higher than 1%. Hu et al. [22] demonstrate that physical layer performance metrics are useful in detecting and predicting network outages that can affect customer experience. Schulman and Spring [26] employ ICMP echo request packets to measure how weather affects the availability of broadband networks. Padmanabhan et al. [25] point out that the last-mile is often the bottleneck by analyzing the end-to-end client-server traffic. Their results indicate that approximately 75% packet loss occur in the last-mile networks. The results presented by Sundaresan et al. [27] support this statement by analyzing the RTT of different TCP traffic. Sundaresan et al. [29] also show the home wireless network is the main bottleneck when a user's access link speed exceeds about 20 Mbps. However, Bajpai et al. [8] measured the last-mile latency in the US and Europe, showing that the last-mile latency is stable over time, which are inconsistent with the observations made by [25,27,29]. Fontugne et al. [17] investigate the last-mile latency among 646 ASes, and find that nearly 10% of the ASes presenting persistent last-mile congestion.

Backbone Packet Loss: Apart from the last-mile networks, researchers have also measured the performance of backbone networks. Ghobadi and Mahajan [19] measure the performance metrics from the optical layer in a large backbone network. Their work shows that one of the optical layer performance metrics, SNR can be used to predict network outages that are not visible to the IP layer. Markopoulou et al. [23] send probes over 43 paths in 7 ISPs to measure the latency and packet loss in the continental US, showing that the packet loss rates for all measured paths are less than 0.26%.

Datacenter Packet Loss: Benson et al. [10] measure packet loss in datacenter networks and show that the packet loss mostly occurs at edge links with low average utilization, indicating the primary cause of packet loss in datacenter networks is momentary spikes. Zhang et al. [31] show most of the packet loss in datacenter networks occur in ToR switches. Both of the studies focus on packet loss in the IP layer. Zhuo et al. [32] show corrupted optical links in datacenter networks introduce a high packet loss rate and the rate of link corruption is not correlated with the link's utilization.

Summary: Different from previous work, this work uses physical-layer codeword statistics to characterize packet loss caused by physical-layer transmission errors. It focuses on the last-mile cable broadband networks and complements previous work.

9 Conclusion

As many applications are sensitive to packet loss, continuously monitoring packet loss in a broadband network has attracted much interest from researchers and policymakers. Previous measurement work, including FCC's decade-long MBA project, cannot differentiate congestion-induced packet loss from transmission-error-induced loss.

This work fills in this blank by using physical-layer data contributed by a cable ISP. The data were collected from 77K+ devices spanning 394 HFC network segments in a 16-month period. Using this data, we infer that physical-layer transmission errors could contribute to more than 12%-25% of packet loss in the cable ISPs measured by the MBA project. We show that some HFC network segments suffer from persistent error loss that exceeds 1%. Customers in these network segments do not make more calls than other customers. These findings suggest that network researchers and operators should take into account packet loss caused by physical-layer errors in network measurement, protocol design, and network maintenance tasks.

Acknowledgment

References

[1] Broadband Internet Regulation and Access: Background and Issues. `https://www.everycrsreport.com/reports/RL33542.html`, 2008.

[2] Measuring Broadband America. `https://www.fcc.gov/general/measuring-broadband-america`, 2011.

[3] DOCSIS Codeword Errors And Their Effect on RF Impairments. `http://www.zcorum.com/wp-content/uploads/DOCSIS-Codeword-Errors-Their-Effect-on-RF-Impairments.pdf`, 2013.

[4] Measuring Broadband America (Technical Appendix to the Tenth MBA Report). `https://data.fcc.gov/download/measuring-broadband-america/2020/Technical-Appendix-fixed-2020.pdf`, 2020.

[5] Measuring Broadband America (Validated Data Cleansing, Tenth Report). `https://data.fcc.gov/download/measuring-broadband-america/2020/validated-data-cleansing-sept2019.pdf`, 2020.

[6] IBM Environmental Intelligence Suite: Weather Data APIs. `https://www.ibm.com/products/environmental-intelligence-suite/data-packages`, 2021.

[7] Number of Fixed Broadband Subscribers in the United States from 2010 to 2020. `https://www.statista.com/statistics/217938/number-of-us-broadband-internet-subscribers/#statisticContainer`, 2021.

[8] Vaibhav Bajpai, Steffie Jacob Eravuchira, and Jürgen Schönwälder. Dissecting Last-mile Latency Characteristics. *ACM SIGCOMM Computer Communication Review*, 47:25–34, 2017.

[9] Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. Pearson Correlation Coefficient. In *Noise reduction in speech processing*, pages 1–4. Springer, 2009.

[10] Theophilus Benson, Aditya Akella, and David A Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *ACM IMC*, 2010.

[11] DOCSIS CableLabs. Best Practices and Guidelines, PNM Best Practices: HFC Networks (DOCSIS 3.0). Technical report, CM-GL-PNMP-V03-160725, 2016.

[12] DOCSIS CableLabs. Data-Over-Cable Service Interface Specifications DOCSIS® 3.0 Operations Support System Interface Specification. Technical report, CM-SP-OSSIv3.0-C01-171207, 2017.

[13] DOCSIS CableLabs. Data-Over-Cable Service Interface Specifications DOCSIS® 3.0 Physical Layer Specification. Technical report, CM-SP-PHYv3.0-C01-171207, 2017.

[14] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-based Congestion Control: Measuring Bottleneck Bandwidth and Round-trip Propagation Time. *Queue*, 14(5):20–53, 2016.

[15] Wireline Competition. Restoring Internet Freedom. `https://docs.fcc.gov/public/attachments/FCC-17-166A1.pdf`, 2017.

[16] Marcel Dischinger, Andreas Haeberlen, Krishna P Gummadi, and Stefan Saroiu. Characterizing Residential Broadband Networks. In *ACM IMC*, 2007.

[17] Romain Fontugne, Anant Shah, and Kenjiro Cho. Persistent Last-mile Congestion: Not so Uncommon. In *ACM IMC*, 2020.

[18] Daniel Genin and Jolene Splett. Where in the Internet is Congestion? *arXiv preprint arXiv:1307.3696*, 2013.

[19] Monia Ghobadi and Ratul Mahajan. Optical Layer Failures in A Large Backbone. In *ACM IMC*, 2016.

[20] Sangtae Ha, Injong Rhee, and Lisong Xu. CUBIC: A New TCP-friendly High-speed TCP Variant. *ACM SIGOPS operating systems review*, 42(5):64–74, 2008.

[21] Dennis E Hinkle, William Wiersma, and Stephen G Jurs. *Applied Statistics for the Behavioral Sciences*, volume 663. Houghton Mifflin College Division, 2003.

[22] Jiyao Hu, Zhenyu Zhou, Xiaowei Yang, Jacob Malone, and Jonathan W Williams. CableMon: Improving the Reliability of Cable Broadband Networks via Proactive Network Maintenance. In *USENIX NSDI*, 2020.

[23] Athina Markopoulou, Fouad Tobagi, and Mansour Karam. Loss and Delay Measurements of Internet Backbones. *Computer communications*, 29:1590–1604, 2006.

[24] Ramakrishna Padmanabhan, Aaron Schulman, Dave Levin, and Neil Spring. Residential Links Under the Weather. In *ACM SIGCOMM*. 2019.

[25] Venkata N Padmanabhan, Lili Qiu, and Helen J Wang. Server-based Inference of Internet Link Lossiness. In *IEEE INFOCOM*, 2003.

[26] Aaron Schulman and Neil Spring. Pingin' in the Rain. In *ACM IMC*, pages 19–28, 2011.

[27] Srikanth Sundaresan, Mark Allman, Amogh Dhamdhere, and Kc Claffy. TCP Congestion Signatures. In *ACM IMC*, 2017.

[28] Srikanth Sundaresan, Walter De Donato, Nick Feamster, Renata Teixeira, Sam Crawford, and Antonio Pescapè. Broadband Internet Performance: A View From the Gateway. 41:134–145, 2011.

[29] Srikanth Sundaresan, Nick Feamster, and Renata Teixeira. Home Network or Access Link? Locating Last-mile Downstream Throughput Bottlenecks. In *International Conference on Passive and Active Network Measurement*. Springer, 2016.

[30] Srikanth Sundaresan, Nazanin Magharei, Nick Feamster, and Renata Teixeira. Characterizing and Mitigating Web Performance Bottlenecks in Broadband Access Networks. In *ACM IMC*, 2013.

[31] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. High-resolution Measurement of Data Center Microbursts. In *ACM IMC*, 2017.

[32] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Klaus-Tycho Förster, Arvind Krishnamurthy, and Thomas Anderson. Understanding and Mitigating Packet Corruption in Data Center Networks. In *ACM SIGCOMM*, 2017.

A Raw Data of Codeword Error Rates

This appendix section includes sample figures of the raw codeword data we used for the analysis in this paper. Figure 18 includes codeword error rates of modems under different conditions. We draw these figures using the data collected from 12 modems between July 1st to July 31st. Figure 18(a) to Figure 18(d) show the data collected from modems in unhealthy FNs. They have high codeword error rates as expected. Figure 18(e) to Figure 18(h) show the data collected from modems in alarming FNs, while Figure 18(i) to Figure 18(l) show the data collected from modems in healthy FNs.
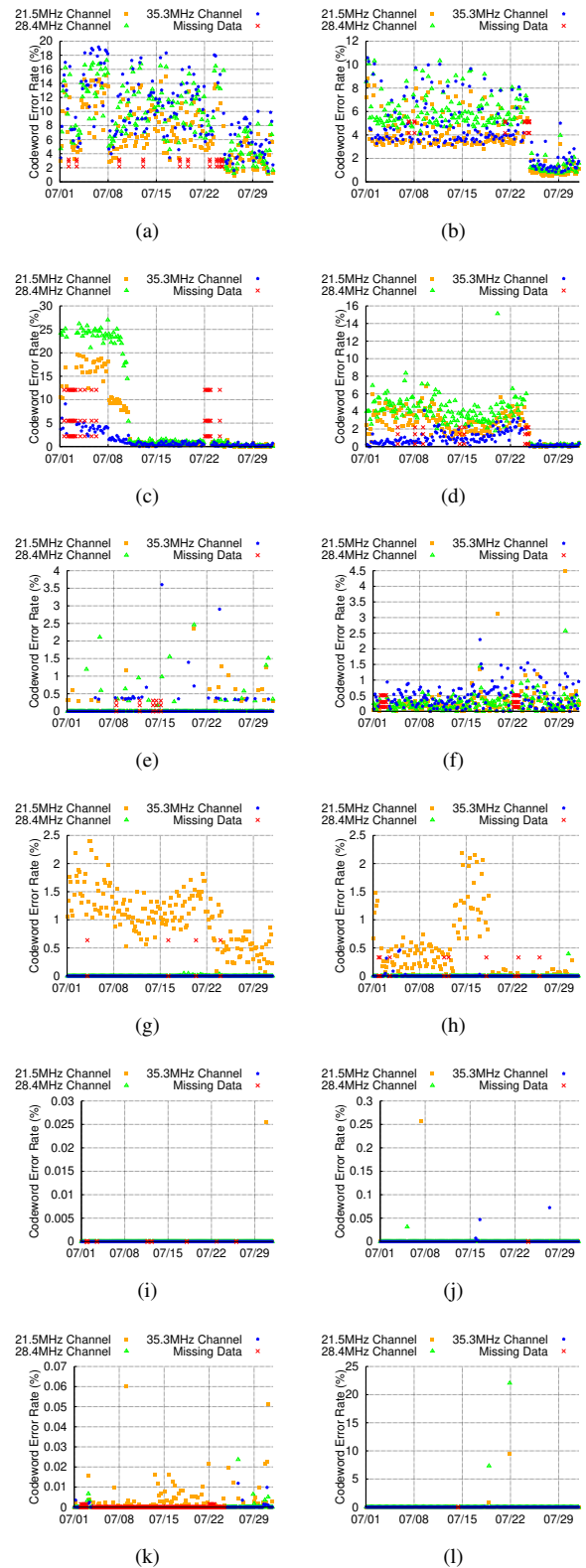
Figure 18: **Raw codeword error rates from sample devices.**

How to diagnose nanosecond network latencies in rich end-host stacks

Roni Haecki[1], Radhika Niranjan Mysore[2], Lalith Suresh[2], Gerd Zellweger[2],
Bo Gan[2], Timothy Merrifield[2], Sujata Banerjee[2], Timothy Roscoe[1]
[1]*ETH Zurich,* [2]*VMware*

Abstract

Low-latency network stacks have brought down network latencies within end-hosts to the *microsecond-regime*. However, end-host profilers have such high overheads that they are useful only to confirm a hypothesis, not to diagnose a problem in the first place. Indeed, every one of twenty low-latency network projects we surveyed rolled their own analysis tools instead of using an existing profiler.

This paper shows how to build a latency diagnosis tool with full-stack coverage and low overhead that can identify, not just confirm, sources of latency in end hosts. The unique measurement methodology reconstructs network-message lifetimes within end hosts with nanosecond precision, by reconciling CPU and NIC hardware profiling traces across multiple time domains (network and CPU). It uncovers unexpected latency sources in both kernel and user-space stacks.

We demonstrate these capabilities by using our implementation, NSight, to systematically identify and remove performance overheads in memcached, reducing 99.9th percentile latency by a factor of 40 from 2.2 ms to 41 µs.

1 Introduction

Operating systems and network stacks are routinely blamed for increasing network latencies. Clearly, we need diagnostic tools to identify sources of latency in end-host stacks. Thankfully, there is no paucity of end-host profilers [1,3,4,6, 13,16,22,25,29,31,37,38,49–51,73,77]. We examined 21 networking projects whose goal was to achieve low latency [2,5,8,11,20,23,26,27,33–36,41,46,48,55,57,58,60,65,69]. Surprisingly, not one of these projects have used these profilers! Instead, all of them design their own handcrafted latency measurement system. This indicates that in spite of the excellent and vast body of prior work, there is no diagnostic tool for *network latencies* introduced at the end host, especially in the microsecond regime. In this paper we present NSight to address this important gap.

Our investigations identify three reasons that existing end-host profilers fail at network latency diagnosis. First, existing profilers fail to capture latency deviations added by the *NIC*, from the point when messages enter (or exit) the NIC to the point that they are received by (or exit) the driver. Many system designs identify these latency deviations to be important [8,24,26,34,35,48,55,58,60,62].
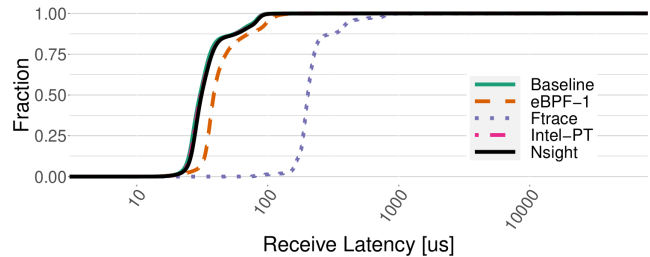
Figure 1: CDF of memcached request receive-latencies with and without profiling. eBPF-1 stands for eBPF probing a single function; Ftrace, Intel-PT and NSight, profile all system functions in the end-host stack. eBPF-1 and Ftrace add variable latencies to functions being profiled that are hard to differentiate from true latency deviations.

Second, their high overheads severely disturb the latency distribution, overwhelming the root causes being pursued. Figure 1 shows how two widely-used profilers, eBPF and Ftrace, add to memcached request-receive latencies, measured from the point requests are received at the NIC to the point they are received by memcached (socket recv). eBPF adds 18-40% overhead while measuring the latency of a single function, while Ftrace adds 298-841% profiling all functions of the end-host software stack. Also shown is the minimal impact of NSight and in the same range as Intel-PT, which is an example of a hardware CPU profiler that NSight builds on.

Third, existing profilers are too heavyweight to apply to the entire stack. A developer diagnosing network latencies must already have a guess of where to look before such a profiler is useful. Blogs [15,28,32,43,45] tell exactly this story: Users determine the parts of the stack that *might* add latency deviations and then use profiling tools to measure latencies in these parts. The unfortunate result is that the latency sources from unexamined parts of the stack are not caught. In addition, important interactions between different parts of the stack go unnoticed [46,68]. For example, profiling the NIC separately from the network software stack hides the impact of NIC deviations on scheduling decisions. When there are large latency deviations at the NIC, a CPU waiting for network messages can idle and go into a lower power state. This increases scheduling latency, and in turn, overall network latency (§7.4 has an example).

Our contribution. We correct these shortcomings by demonstrating the feasibility of a low-overhead, holistic tool called NSight for diagnosing network latency deviations introduced at end hosts. The guiding principle is that *entire lifetimes of network messages within end hosts* must be examined to determine the precise causes of latency deviations. These lifetimes are defined by *all system activity*, not limited to network processing, that impacts messages from the time they enter end hosts to when they exit it. To be useful in the microsecond regime, NSight must reconstruct these lifetimes with nanosecond precision. NSight does so by reconciling timelines of two fine-grained data sources, CPU hardware profiling and NIC hardware timestamps.

This reconstruction is challenging for two reasons. First, the two data sources record time using different hardware clocks. Hardware CPU profiling uses a monotonically increasing clock for capturing precise *intra*-end-host latencies, while NIC and software CPU clocks are often synchronized using PTP [61] to aid *inter*-end-host latency measurements. To align the timestamps in these sources correctly, NSight tracks the conversion between the two time domains during profiling.

Second, CPU hardware profiling does not track the passage of network messages in multi-core systems across kernel cores (which process the message) and application cores. To track this path, NSight captures timestamps and core numbers at the boundary where kernel hands off the message to the application. This boundary is also the point where message processing can move across cores.

Once the lifetimes of messages are constructed, they can be compared to identify anomalous processing that led to their latency deviations. Unfortunately, due to the deep nesting of end-host call stacks, latency deviations in functions introduce deviations in their *parent* functions making them look anomalous too! To reduce ambiguity in attributing root causes to anomalies, NSight traverses the call stacks until it finds functions with latency deviations that cannot be attributed to nested calls.

NSight demonstrates that these techniques are sufficient to overcome the listed challenges, while incurring overheads comparable to hardware profiling (see Figure 1). Due to its low overhead, NSight can be used to diagnose even sub-microsecond increases in network latency at the end host. We describe our use of NSight to profile both the Linux kernel and Mellanox's VMA [48], a user-space network stack. On these stacks we describe how we profile unmodified applications `memcached` and `redis`. We dive deeply into a detailed case study of `memcached`, the application of choice in several low-latency end-host stack papers [11, 35, 36, 39, 55, 58, 60, 70, 71], to prove the diagnosis capability of NSight. In the case study NSight systematically narrows down the bottlenecks of the Linux stack, as we mitigate 99.9th percentile `memcached` tail latency by over 40x - from 2.2 ms to 51 μs. When one of the mitigations increases latencies below the median by as much as 11 μs, NSight helps identify the reasons for this increase.

2   Background and related work

The vast body of profiling and diagnostic tools fall broadly into two groups: fault diagnosis tools ([10, 64, 74]) and performance diagnosis tools ([21, 75, 76]). Some diagnose problems *within* end-hosts ([1, 3, 4]). Others help in distributed settings ([19, 21, 68]). NSight is a performance diagnosis tool for end-hosts. Its focus is on *latency* diagnosis. We focus on software- and hardware- based tools built for latency diagnosis below.

Software-based end-host tools. Software-based diagnostic tools are built on top of profiling data sources such as probes (Uprobes [17], Kprobes [30]), kernel tracepoints, performance counters, and software tracing. The data source determines the overheads and insights that can be drawn from the tool.

Tools that depend on probes (LTTng [13], eBPF [1]) and kernel tracepoints (dtrace [51]) allow users to instrument functions of interest. Users of these tools must already know where to look, making them less suited for latency diagnosis than tools that depend on software tracing (Ftrace [3]). Tools based on performance counters (`perf stat` [4]) do not capture outlier latency events (due to aggregation) making them unsuited for tail latency diagnosis in particular.

Unlike NSight, all of these tools miss NIC delays altogether and have much higher overheads than hardware profilers.

Hardware-based end-host tools. CPU profilers ([12, 40, 53]) record software function `call` and `return` times, their core number and process names at processor speeds. The timestamps help measure software latencies with nanosecond precision and low overhead. NSight, like `perf` and `VTune` [25], uses CPU profilers to track software function latencies.

Most NICs support hardware timestamping [14] for network packets. The timestamps can be retrieved per-message using standard Linux socket calls, to determine network latencies across and within end-hosts. NSight uses NIC timestamps to track entry and exit of network messages during profiling.

Unlike NSight, these tools cannot identify the sources of network latencies in end-hosts by themselves or when combined with one another. Instead they require the user to make a conjecture that the tools can help confirm (See §2.1 and §4).

Distributed tools. Even though NSight is designed for network latency diagnosis *within* end-hosts rather than in distributed settings, some similarities and differences are worth noting. Like NSight, many diagnostic tools for distributed systems [7, 9, 18, 19, 44, 54, 68, 72] reconstruct the path of messages but in distributed settings. They do not capture network latency sources at end-hosts but capture other latency sources like packet drops, routing issues and workload spikes.

Inband network telemetry (INT) [56] is a mechanism to probe specific points in network dataplanes. In software, these probes are expensive [67] and do not cover many points in end-host stacks, like the OS stack, that we do with NSight. Therefore they can only confirm causes of latency, not identify them. On the other hand, INT works with all programmable network hardware while NSight is relevant only for end-hosts.
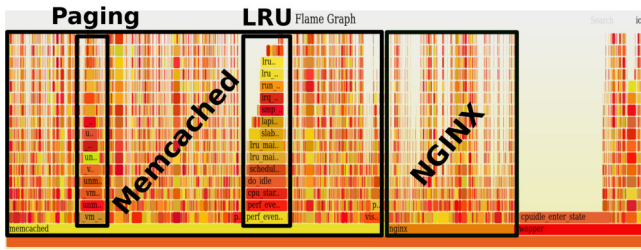
Figure 2: A flame graph is not very useful for diagnosing latency problems because it only highlights common events in system activity. It leaves the identification of events that are deviations from the norm to the user.

2.1 Diagnosing network delays at end-hosts

We now describe common network latency debugging practices at end-hosts using existing tools (blogs [15, 28, 32, 43, 45]). The process usually begins with tools that summarize system activity, like `perf` [4], `SystemTap` [59], unless developers already have deep knowledge of the application of interest [32]. These tools help identify the parts of the stack that are most active and *might* impact network latency.

For example, we can generate flame graphs like Figure 2 with `perf`. The flame graph shows two applications, `memcached` and `NGINX` both running on the same machine. It is natural to ask whether they interfere to cause network delays [20, 27, 55]. To verify, developers can isolate applications, or turn on software tracing, like `Ftrace`, and visualize network send/receive paths using tools like `KernelShark` [66].

Figure 2 also shows `memcached` LRU cache maintenance and paging activity to be high. Even though unrelated, these *can* delay network activity if scheduled during network processing. To verify, developers can use tools like eBPF or Ftrace to track periods of such activity and correlate them with periods of high network latencies. To verify delays *on* network processing paths, these tools can generate latency distribution graphs for the functions of interest [45].

These steps alone might be insufficient [15]. System activity summaries like Figure 2 do not show problems like NIC delays, scheduling bottlenecks, or head-of-line blocking that can slow down network processing. Finding these problems requires tracking specific system/NIC performance counters [4, 52] or handcrafted measurements [35, 39].

3 Using NSight for diagnosis

We now describe a typical debugging experience with NSight. Let us suppose users are running `memcached` on Linux and see large tail latencies despite light query loads. To use NSight to quickly diagnose the cause of poor performance, the user first turns on NSight profiling for a second. Once NSight analyzes the profiling data, the user inspects the initial result, Figure 3. This *balloon plot* shows the latencies of all `memcached` requests profiled and the top causes of the slow-
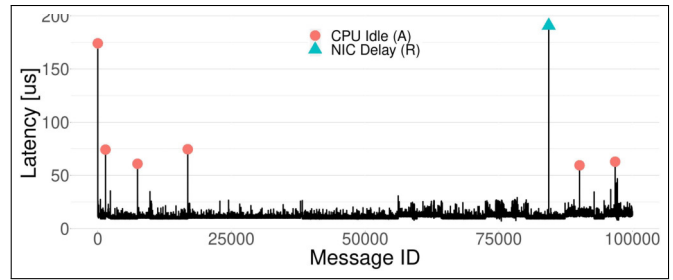
Figure 3: Balloon plots summarize message latencies(Y-axis) with the top causes for latency deviations (legend) mapped to corresponding messages using similar balloons.The balloons are in Message ID order (X-axis) to identify bunching together of balloons that indicates a single underlying cause. If they are spaced apart, there are likely multiple independent reasons that need to be addressed separately.
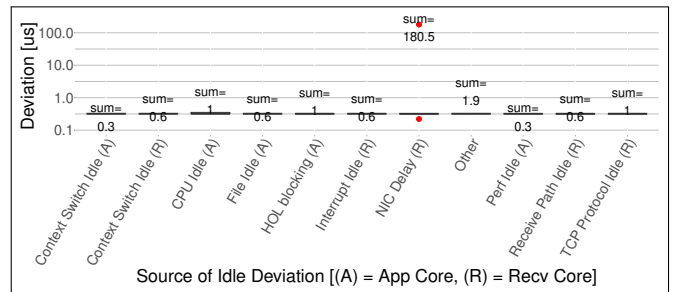
Figure 4: A zoomed-in box plot of causes for CPU idling that impacted the slowest message. This presentation is valuable for identifying why the cores were idling (X-axis), and the latency deviation due to those causes (Y-axis). Both the application (*A*) and receive cores (*R*) idle, for reasons ranging from waiting during a context switch or file access, with NIC interrupt coalescing, *NIC delay (R)*, causing the largest wait.

est (tail) requests. The user notices that the tail requests are largely slowed down by idling CPU cores, *CPU Idle (A)*, where A stands for application-core on which `memcached` runs. The slowest request is also slowed down by core idling due to NIC interrupt coalescing delay, *NIC Delay (R)*, where R stands for the core on which the kernel receives the request.

From this result, the user can drill down into the presented causes for tail latencies. For instance, the user cross-checks why the cores were idling when processing the slowest request. Figure 4 shows that, among various causes, the largest latency deviation was due to NIC interrupt coalescing.

The user could also manually verify this diagnosis using Figure 5, a detailed timeline of all system-activity that impacted the slowest message. Eyeballing the presentation, the user confirms that there is a gap in system activity *before* the message is processed but long *after* the message is received at the NIC, showing that interrupt coalescing delays slowed down the tail request by as much as 180 μs.
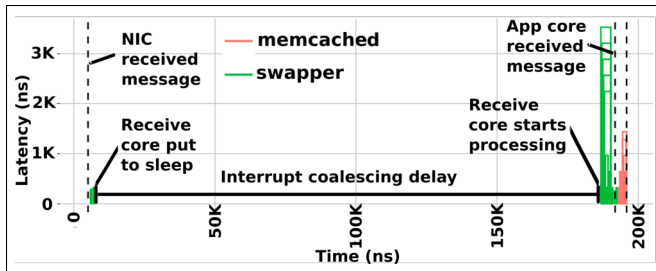
Figure 5: NSight's presentation of processing timeline of the slowest message. The presentation tracks *system activity* from the time the message was received at the NIC (leftmost dashed vertical line) to the time it was received at the application (rightmost dashed vertical line). The reader can ignore the Y-axis for now; a full explanation of this presentation is available in Figure 8. Gaps in system activity indicate CPU idling.

4 Challenges and key ideas

Why is latency diagnosis hard? To derive the precise causes of message slowdowns within end-hosts, we must profile *network-message lifetimes*, like Figure 5. These lifetimes capture *all system activity (G1)* that slowdown message processing, whether it is network stack activity or something unrelated, from the point a network message enters (or exits) the end-host to the point it is received (or sent) by the application. To work with low-latency stacks, we must capture these lifetimes *automatically (G2)*, with *nanosecond-precision (G3)* and *low overhead (G4)*. We must then analyze these message lifetimes to identify system activity that slows messages down and their precise impact, as shown in Figure 3.

4.1 Profiling network-message lifetimes

Unfortunately, the task of capturing network-message lifetimes is difficult, because there is not *one* system component that processes network messages. Rather, in most modern OSes [42, 47, 55], network messages are processed across the NIC and one or multiple CPU cores. Capturing message lifetimes that span these devices is challenging for two reasons.

Challenge 1 (C1): **NIC, CPU profiling, and software timestamps come from independently *changing* clocks.** Figure 6 illustrates this problem. To construct message lifetimes NIC, CPU profiling, and software clocks must align at all times, but they do not. NIC and CPU profiling clocks drift independently of each other. This is not a problem for CPU profiling, because it uses the hardware clock to report nanosecond-granularity function latencies *within* an end-host.

NIC timestamps are used to measure *both* inter-host and intra-host message latencies. To aid latency measurement across multiple time-domains, socket libraries convert NIC timestamps to software timestamps *during profiling*, with the help of conversion parameters calculated by software synchronization mechanisms like `phc2sys` on Linux.
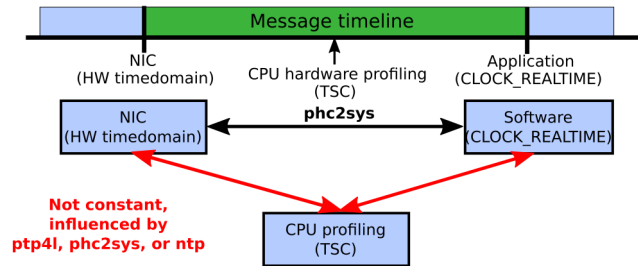
Figure 6: Constructing message lifetimes from timestamps taken on NIC and CPU, whose clocks are independent, is hard because the timestamps do not line up into a single timeline. We overcome this challenge by monitoring the relationship between CPU profiling clock and NIC/Software clocks (shown in red arrows) *during* profiling with low *latency*-overhead.

Software clocks across end-hosts are usually synchronized using protocols like `ptp`. These synchronization mechanisms constantly readjust software clocks and change their relationship with the CPU profiling clock, under the hood. These changes pose a problem for constructing message lifetimes. ***Key idea 1, Time reconciliation:*** The ideal solution to this problem would be to modify time synchronization protocols to expose clock changes to profilers like `perf`. Profilers could then present all measurements based on the software clock to simplify system-wide latency measurements.

We do not want to refactor the entire software stack, so we use a simpler but more expensive workaround. *After* time synchronization protocols change the software clock, the kernel recalculates the conversion between CPU profiling clock and software clock. By exposing these conversion parameters to user-space using virtual dynamic shared object (vDSO) mechanisms, we can poll them from a user-thread for the duration of profiling. The goal is to capture *every* change to the software clock to have the most accurate mapping between the clocks at all points in time. A log of the conversion parameters helps reconcile CPU profile timestamps to software-clock domain *post* profiling. This method burns an entire core for the user-thread, but it adds no latency overhead *during* profiling because the user thread is isolated from the rest of the system.

Challenge 2 (C2): **There is no support to track network-message lifetimes *within* host software stacks.** Software profiling tools can only track network messages *within* the network stack, missing other system activity that slows down message processing. This is because CPU architecture and operating systems treat network messages like regular data structures, as objects in memory. Consequently, network messages have no relevance outside of network stack functions.

On the other hand, CPU profiling tracks all system activity but not the message lifetimes within which such system activity occurs. Figure 7 illustrates this problem. The two rectangular boxes labeled Core **X** and Core **Y** show the timeline of system activity captured by CPU profiling at these two
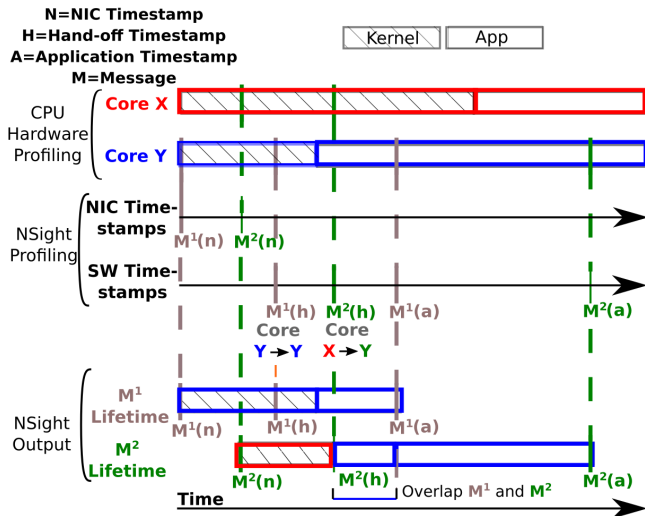
Figure 7: We cannot reconstruct message lifetimes using CPU profiling and NIC timestamps alone. Instead, we place CPU-profile timelines in the context of each message by collecting *per-message* NIC timestamps ($M^1(n)$, $M^2(n)$), Software timestamps at application ($M^1(a)$, $M^2(a)$), and cross-core hand-off timestamps along with the core information ($M^1(h)$ Core $\mathbf{Y} \rightarrow \mathbf{Y}$, $M^2(h)$ Core $\mathbf{X} \rightarrow \mathbf{Y}$). The superscript of each timestamp corresponds to the message ID assigned by NSight. With this context, we see that M^1 is received at NIC at $M^1(n)$ and processed on core $\mathbf{Y}$ through $M^1(h)$, till it is received by App at $M^1(a)$. M^2 is processed on core $\mathbf{X}$ and then handed-off to the App on core $\mathbf{Y}$ at $M^2(h)$. At Core $\mathbf{Y}$, M^2 waits for M^1 to be processed before it is received by the App at $M^2(a)$.

cores. We only see the process contexts (kernel, application) and individual functions (not shown), but not messages.

Key idea 2, Message profiling: To construct message lifetimes, we augment CPU profiling with *per-message* timestamps and core numbers. Figure 7 shows the information collected by NSight profiling to construct two message lifetimes, M^1 and M^2. For each message (for example, M^2), we seek to get three timestamps: a NIC timestamp ($M^2(n)$), a core hand-off timestamp ($M^2(h)$), and an application timestamp ($M^2(a)$). The NIC and application timestamps allow us to capture the start and end of a message's lifetime on the end-host; for instance, M^2's lifetime is from $M^2(n)$ to $M^2(a)$. The core hand-off timestamp ($M^2(h)$) along with core information (Core $\mathbf{X} \rightarrow \mathbf{Y}$), helps identify the *per-core* system activity that a message encounters in its lifetime; for example, M^2 is processed on Core $\mathbf{X}$ between $M^2(n)$ to $M^2(h)$, and then on Core $\mathbf{Y}$ between $M^2(h)$ to $M^2(a)$. Crucially, these timestamps are sufficient to produce a single timeline of all system activity related to the processing of a message.

We now explain how these timestamps can be obtained and how they suffice to reconstruct detailed message lifetimes. The per-message NIC hardware-timestamps and appli-
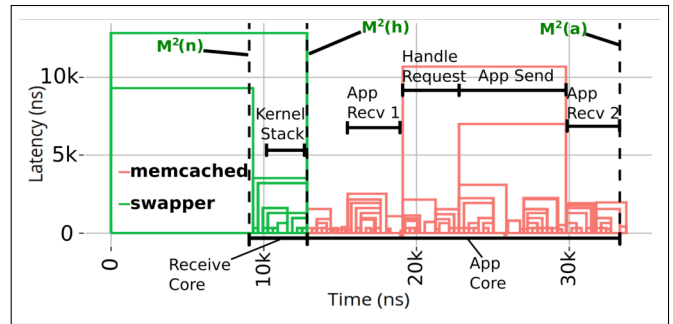
Figure 8: NSight presentation of message lifetime of M^2. The X-axis shows time in nanoseconds and captures message processing latency from $M^2(n)$ to $M^2(a)$. System activity is captured as nested 'boxes' on the timeline. Each box represents a function traced by CPU profiling. The left and right vertical boundaries of each box corresponds to the function `call` and `return` timestamps. The Y-axis shows individual function latencies in nanoseconds. The horizontal black lines and annotations (e.g., `App Recv`) are added for clarity.

cation software-timestamps can be taken at the application send/receive operations. They help *order* messages, so we can assign message IDs, M^1 and M^2. Messages *sent* from end-hosts are ordered by their application timestamps. Messages *received* at end-hosts, like M^1 and M^2, are ordered by NIC timestamps, $M^1(n)$ and $M^2(n)$. The per-message *hand-off* timestamps and core information can be collected at points where messages cross software-processing and core boundaries. In kernel network stacks, there is one boundary where messages are handed-off between kernel and application, like `sock_def_readable` in Linux. User-space network stacks can have similar boundaries [47, 55] while others do not [26, 48, 58].

In our example in Figure 7, there is an overlap between lifetimes of M^1 and M^2 on core $\mathbf{Y}$ from $M^2(h)$ to $M^1(a)$. M^2 has to wait for the application on core $\mathbf{Y}$ to complete processing M^1 before it can be processed. This overlap shows inter-message interference or head-of-line blocking. We can similarly detect unrelated system activity or application interference that appear in message lifetimes.

Figure 8 describes NSight's presentation of M^2. Between timestamps $M^2(n)$ and $M^2(h)$, M^2 is processed by core $\mathbf{X}$ (`Receive core`). After $M^2(h)$, M^2 is processed by core $\mathbf{Y}$ (`App core`) where M^2 waits for the application, `memcached`, to process M^1 (`App Recv 1` to `App Send`), after which M^2 is received (`App Recv 2`).

4.2 Diagnosing high message latencies

Profiling even for brief periods of time, will leave us with hundreds of thousands of message lifetimes. To diagnose why some message lifetimes are longer, like we did in Figure 3, we must identify *anomalous* system activity that slow down those
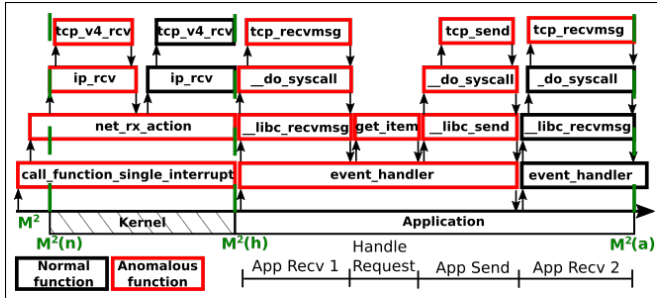
Figure 9: M^2-message-lifetime shown in greater detail by zooming into part of the call stack. When functions within M^2 are compared to those in M^1, many functions shown in red appear anomalous. For instance, `call_function_single_interrupt` and its nested functions appear to take longer or occur more frequently. The *cause* for these deviations is that `ip_rcv` has been called twice, once for receiving M^1 and a second time for receiving M^2 due to head-of-line blocking.

messages relative to others, by comparing their lifetimes. Two typical types of anomalies in message lifetimes are functions that take longer and functions that occur more frequently in some lifetimes compared to others; for example, App Recv occurs twice in M^2 in Figure 8. Deeply nested end-host stacks can result in nesting of anomalies of different types, leading to a third challenge.

Challenge 3 (C3): **Due to nesting, the same latency deviation can be explained by multiple anomalies.** Figure 9 explains this problem. When compared to M^1, most functions in M^2 appear anomalous! Some take longer, like the first invocation of `event_handler`; others appear more often, like `__libc_recvmsg`; and the rest are unexpected, like `__libc_send`, a *send* in the middle of *receiving* M^2.

Key idea 3, Anomaly disambiguation: To reduce ambiguities from nesting, we only report anomalous functions that cannot already be *explained* by their nested functions. To determine whether an anomalous function is explained by nested anomalies, we use a heuristic. If the nested anomalies together account for more than 80% of the latency deviation of the parent function (see §5.2 for why), we conclude that the nested anomalies explain the parent anomaly and omit the parent anomaly as a reason for deviation.

Figure 9 describes how the latency deviation in `call_function_single_interrupt` is accounted for by a nested anomalous `ip_rcv` call. Therefore, `ip_rcv` is listed as a root cause but not `call_function_single_interrupt`.

5 Design and implementation

In this section, we describe how the three key ideas from §4 realize the goal of network latency diagnosis in microsecond-regimes. NSight is composed of two subsystems. The first is a profiling subsystem, that tracks broad CPU activity and
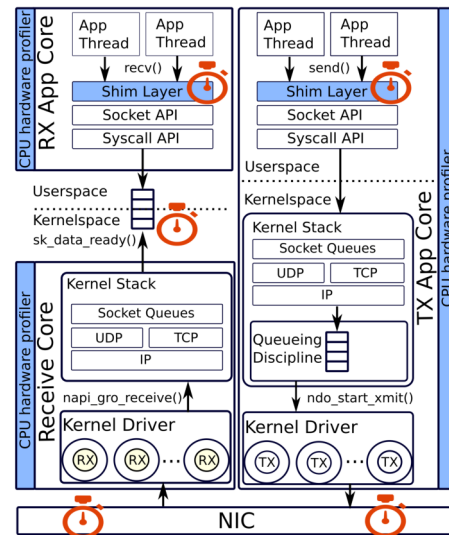
Figure 10: NSight profiling in Linux. The unshaded parts of this figure are the scope of activity that we want to profile and automatically diagnose over. We run CPU profiling on all cores to collect system activity *(G1)* and identify anomalies. To establish *causality* linking observations from the CPU profiler to messages, we collect timestamps and core information for each message on their way in and out of the system and across cores with the shim layer.

passage of messages through end-host stack (§4.1, *Message profiling*). It also tracks the relationship between NIC, CPU profiling and software time-domains so that the observations from different devices can be aligned into a single timeline (§4.1, *Time reconciliation*). Put together, it is responsible for capturing *all system activity (G1)* during the profiling period with *nanosecond-precision (G3)* and *low-overhead (G4)*. The second is an analysis subsystem that reconstructs network-message lifetimes and diagnoses network latency deviations within them. To do so, it analyses anomalous system activity in network-message lifetimes, identifies root causes, and attributes precise deviations to these root causes (§4.2, *Anomaly disambiguation*). The profiling and analysis subsystems together *automate (G2)* network latency diagnosis.

5.1 NSight profiler

Figure 10 explains how we get broad information from CPU hardware profiling *(G1)* and combine it with per-message profiling information from a shim layer to establish a causal link between system behavior and message lifetimes (§4.1, *Message profiling*). Typical CPU profiles collect system activity at function granularity and list *all* function `call` and `return` times, their execution context and core numbers with nanosecond granularity *(G3)*. Independent of the CPU profiler, the shim layer intercepts messages, irrespective of the application, to collect timestamps at three points in the stack, shown

in Figure 10, as metadata for each message. Core numbers are collected at the kernel–userspace boundary. For socket-based stacks, the shim layer also collects socket file descriptor numbers to detect head-of-line blocking that arises when application threads are multiplexed across multiple sockets. We quantify overheads from profiling in Section 6 *(G4)*.

To reconcile independent observations from the CPU profiler and the shim layer into a single timeline, §4.1, ***Time reconciliation*** polls the conversion between the clocks for the duration of profiling from a user-thread. To construct message lifetimes accurately, the user-thread must detect *all conversion changes* and align the observations at all points in time. We quantify the accuracy of this scheme in §6 *(G3)*.

Profiler implementation and deployment. The shim layer is implemented using standard Linux and socket APIs, with 1055 lines of C code. It can be dynamically linked by *unmodified* applications using `LD_PRELOAD`; statically linked applications might need to be modified, however.

We extend the Linux NIC timestamp framework to obtain timestamps at the kernel–userspace message hand-off boundary by patching 40 lines of the 5.4 Linux kernel (§4.1, **C2**). VMA stack has no such boundary and needs no modification.

NSight is built on top of the Intel-PT CPU profiler, whose traces are available through `perf`. `perf` exports Intel-PT timestamps from `TSC` to `sched_clock` timedomain. We modify 443 lines in `perf` to use the time reconciliation parameters and export CPU profiler timestamps directly in software time domain (`CLOCK_REALTIME`). The thread that polls these parameters is implemented in 363 lines of C code (§4.1, **C1**).

§7 shows how NSight is effective even when users turn it on for only a few seconds at a time. To capture random events across time, users must turn on NSight repeatedly. We have ambitions of using NSight for continuous profiling, but the current buffering implementation in Intel-PT limits such use.

5.2 NSight analysis

The analysis subsystem consists of three parts. The first part reconstructs message lifetimes from profiling data (§4.1, ***Message profiling***). The second part detects anomalous system activity during these lifetimes. The third part sifts through the anomalies to identify root causes for latency deviations within message lifetimes (§4.2, ***Anomaly disambiguation***, *G2*).

Identifying anomalous system activity. In §4.2 we mentioned three types of anomalous system activity that contribute to latency deviations: functions that take longer, functions that are called more frequently, and unexpected functions that show up in some message lifetimes compared to others.

There are three other classes of anomalies that slow down message lifetimes. The first class consists of *entire program contexts* that are unexpected but show up in message lifetimes as a result of scheduling decisions or interrupts. The second class is defined by the *absence of system activity* that is seen when the CPU is idling. This anomaly occurs when the CPU
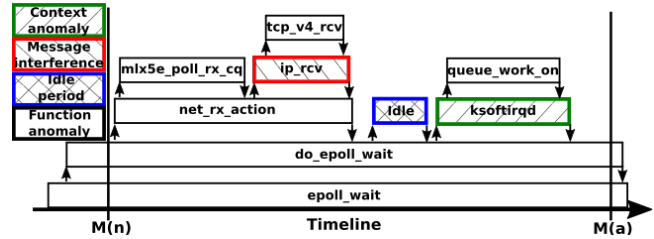
Figure 11: Message lifetime with a richer set of nested anomalies. {`mlx5e_poll_rx_cq`, `ip_recv`, `Idle`, `ksoftirqd`} are sufficient to explain the causes of latency deviations seen in this nested stack (§4.2, ***Anomaly disambiguation***).

waits for an event, like a memory read due to a cache miss, and is often indicative of resource bottlenecks. The third class is *cross-message interference* in the network stack.

The algorithm that classifies system activity within message lifetimes as anomalous or normal runs in four phases. ***The first phase*** compares message lifetimes belonging to the same application and identifies *unexpected program contexts* that occur in a minority of message lifetimes. OS and application interference is usually identified in this phase. ***The second phase*** identifies *gaps in system activity* that are not associated with any function or program context, like Figure 5. It also identifies the cause for each gap from the last function call *preceding* the gap. Figure 4 is an example result. ***The third phase*** compares message lifetimes belonging to the same application and identifies individual *functions* that are slower, more frequently called, or unexpected. Figure 8 presented several anomalous functions identified in `M2` lifetime. Error handling code paths, like `TCP` retransmission, and application bottlenecks are usually identified in this phase. ***The final phase*** identifies overlaps *between* messages, similar to how we detect overlap between `M1` and `M2` in §4.1, ***Message profiling***. This phase detects cross-application network interference when the overlap is between messages belonging to different applications, and head-of-line blocking when the overlap is between messages of the same application.

Attributing root causes to anomalies. §4.2, **C3** discussed how nesting ambiguates latency-deviation attribution. Often, only one of the anomalies in a deeply nested stack is sufficient to *explain the latency deviation due to the stack and identify corrective action*. In such cases, listing the nested anomalies as root causes only ambiguates diagnosis.

We now revisit this challenge in the context of an example in Figure 11 that has an anomalous program context (`ksoftirqd`) and cross-application message interference (`ip_rcv`). It is unnecessary to flag nested anomalies *within* anomalous program contexts or cross-application message interference because the corrective action to address the deviation is clear: the scheduling policy must be revisited to avoid these anomalies. Therefore, the algorithm will not flag `queue_work_on` or `tcp_v4_rcv` even if they are anomalous.

§4.2, ***Anomaly disambiguation*** described how we do not report anomalous parent functions if 80% percent of their latency deviation is already explained by the nested anomalies. When we report nested anomalies instead of their parent anomaly, we will not be able to explain some portion of the latency deviation. This is due to the fact that nested function latencies seldom make up 100% of the parent function latencies. We define the fraction of the latency deviation explained by NSight diagnosis as its *coverage*. For example, if a parent function takes 1000 ns longer than normal, and nested function takes 900 ns longer than normal, reporting only the nested function as root cause will result in 90% coverage. We evaluate NSight's coverage in §6.

Analysis implementation. The algorithms in the analysis subsystem are implemented with 2189 lines of R code. The visualisations comprise 1768 lines of R.

The precision of Intel-PT is limited because it profiles in *batches*, a few CPU cycles at a time. Latencies of functions whose `call` and `return` times are within the same batch can be under-reported to take no time at all! On the other hand, small latencies, corresponding to the batch granularity, can be added between batches, giving the appearance of an idling CPU. Such under-reporting and bogus gaps in system activity obfuscates anomaly detection. NSight analysis algorithms therefore ignore differences in latency smaller than the batch granularity (up to 322 ns in our system).

Instead of directly reporting anomalous functions such as `ksoftirqd` or `ip_recv` that are hard to interpret as root causes, we categorize them by functionality and report a single root cause, like `OS threads` or `Receive processing`. Each category is suggestive of a class of corrective actions that apply to all functions in that category. Table 1 shows 4 examples of categories used in this paper.

The process of categorization is partly automated. For example, process contexts are automatically derived from `perf`. We categorize 1350 functions in Linux by hand, using the contextual information from their names in a user configurable CSV file. We also introduce head-of-line blocking as a category. To do so, we automate summarization of all functions that are executed when processing messages with *minimum latency*. When these functions occur more frequently or their latencies deviate (for example, NIC interrupt processing takes longer) in message lifetimes of the same application, NSight shows head-of-line blocking as one of the root causes. We verify the latency deviations attributed to head-of-line blocking by cross-referencing message lifetimes to identify overlaps.

6   Evaluation

Our work on NSight is motivated by the paucity of *full-stack*, *lightweight* and *high-fidelity* network latency diagnosis tools that can be used to diagnose latencies in the microsecond regime. In this section, we examine to what extent the ideas in this paper address this gap.

Category	Anomalies
`NGINX` head-of-line blocking	`NGINX` process context
	Functions that were involved in processing the fastest network messages.
OS Threads	`ksoftirqd`, `kthreadd`, `kworker`, `swapper`
Receive Processing	Kernel/Driver packet processing (e.g. `ip_rcv`, `net_rx_action`, `mlx5e_poll_rx_cq`)

Table 1: Category examples and corresponding anomalies

To provide *full-stack* visibility, §4.1, ***Time reconciliation*** aligns observations from the CPU profiler and shim layer that use independent clocks. §6.1 examines the accuracy of time reconciliation which is crucial for tracking causality. To be *lightweight*, §4.1, ***Message profiling*** relies on hardware profiling while introducing software profiling *latency*-overheads at a few points in the end-host stack. §6.2 examines the overheads of this profiling scheme for two reasons; first, to determine if it can produce reliable diagnosis *despite* the overheads and second, if NSight can be used in production. Finally to be *high-fidelity* yet unambiguous, §4.2, ***Anomaly disambiguation*** only reports a subset of anomalies that can explain a majority of latency deviations in messages as root causes. §6.3 examines the extent to which this technique is successful especially in the microsecond regime.

6.1   Time reconciliation correctness

Software clock changes make reconciling system activity, captured by CPU profilers, and message lifetimes, captured by shim layer, hard. To align system activity and message lifetimes correctly, §4.1, ***Time reconciliation*** must capture *all* software-clock change events and use the correct conversion between CPU profiling and software clocks at all points in time. If old or incorrect conversion is used, the system activity and message lifetimes will be incorrectly lined up, introducing spurious or missing system events in message lifetimes.

To test the accuracy of §4.1, ***Time reconciliation*** , we design a benchmark that continuously captures CPU profiling (`TSC`) and software timestamps(`CLOCK_REALTIME`) one after the other. When we use the conversion parameters captured by NSight's user-space thread to convert all the CPU profiling timestamps to software timestamps, we expect to construct a linear timeline from the interspersed converted and measured software timestamps; that is, consecutive timestamps must *always advance* in time. If the old or incorrect conversion parameters are used in any time window, we detect a deviation from the linear timeline, showing that events observed across CPU profiling and the shim layer will be reordered.

Across multiple runs on different machines, over 10,000,000 conversions for each run, the consecutive converted and measured software timestamps always advance in time, never deviating. In each run, we observe that the software clock changes 4000+ times. This shows that the proposed time reconciliation maintains the ordering of events in message lifetimes even with software clock changes. We note

All numbers in μs. h = high load, l = low load				
Tool	median (h)	99.9th (h)	median (l)	99.9th (l)
Baseline	30.3	112.6	10	14.4
Intel-PT	30.8 (2%)	120.4 (7%)	10.6 (5%)	15.3 (6%)
NSight	31.1 (3%)	132.8 (18%)	11 (10%)	16.2 (12%)
eBPF-1	38.6 (27%)	157.6 (40%)	11.8 (18%)	17.3 (20%)
eBPF-2	41.8 (38%)	165 (46%)	13.2 (31%)	18.6 (29%)
eBPF-4	51.9 (71%)	556 (393%)	14.1 (41%)	19.4 (35%)
eBPF-8	59.1 (95%)	565 (402%)	15.5 (54%)	21 (45%)
Ftrace	201.8 (565%)	1060 (841%)	40.1 (298%)	66.4 (359%)

Table 2: Overhead of profiling tools on median and tail measurements. eBPF-*n* is eBPF used to profile *n* functions.

that a stronger test for time reconciliation, in which we take two timestamps *at the same time* and test their equivalence is impossible. Capturing timestamps takes time, and is the reason for a majority of NSight's profiling overheads.

6.2 Message profiling overheads

Latency overheads of profiling tools can perturb message lifetimes and obfuscate latency *diagnosis*. This is why these tools are only used for confirming hypotheses rather than diagnosis [15, 28, 32, 43, 45]. Even though NSight perturbs message lifetimes, it does not share this challenge because NSight is built on top of CPU profiling which *profiles NSight itself*! When NSight profiling slows down message lifetimes, it will show up in system activity captured by §4.1, ***Message profiling*** and §4.2, ***Anomaly disambiguation*** will detect it as the root cause. *Perf Idle (A)* in Figure 4 is an example where the CPU idles during NSight profiling introducing latencies.

Latency overheads of profiling tools also inhibit their use for *latency measurement in production environments*. To be useful in the microsecond-regime, we expect that this overhead should not be beyond a few microseconds.

To evaluate the latency overhead of NSight, we run a benchmark using memcached and record the receive latency of requests, measured from the time they arrive at the NIC to when they are received by memcached; this is the baseline measurement. A peak 4 core load on our system is 500k requests per second (rps); for this experiment we measured overheads across two loads on 4 cores, a low load (40k rps, 8% of peak), and a high load (300k rps, 60% of peak).

Table 2 shows the result of the experiment. Intel PT adds 2-7% over all measurements; Gathering software timestamps and core information with NSight adds another 1-11% totaling 3-18% overhead. This shows that NSight can be turned on in production for brief periods of time without perturbing median latencies by more than a few microseconds.

In contrast, as Figure 1 showed, profiling even *a single function call*, like __sys_recvmsg, using eBPF (eBPF-1) adds more overhead than NSight. As we increase the number of functions calls profiled with eBPF, its overhead increases (See eBPF-2, eBPF-4 and eBPF-8). Over all experiments, eBPF-1 adds 18-40% latency overhead. Ftrace, that profiles the full

stack like NSight, has the largest impact on performance (up to 841% overhead). The high latency overheads of these tools make them impractical for use in production environments.

The current design for §4.1, ***Time reconciliation*** burns an entire core for the user-thread. This overhead can be prohibitive in production environments. Thankfully, the overhead is avoidable because software clocks change relatively slowly, once every 4 ms in Linux. If the software clock changes are tracked directly from the time synchronization protocol, there is no need for an extra core during profiling.

6.3 Coverage after anomaly disambiguation

For high-fidelity, latency diagnosis must be able to report root causes for all latency deviations seen in message lifetimes, even if the deviations are in the order of microseconds. We measure fidelity in terms of coverage, defined as the latency deviation explained by diagnosis as a fraction of overall latency deviation. However, §4.2, ***Anomaly disambiguation*** leads to less than 100% coverage. Trading off some coverage for getting rid of false negatives is still reasonable because NSight can be used *iteratively* to improve coverage; in each iteration the most conspicuous root causes remaining can be discovered with NSight and corrected for (See next section).

By not reporting parent anomalies, when 80% or more of their deviation is explained by their nested anomalies, we get a minimum coverage of 69% and a median coverage of 96% across all experiments and all observed message-latencies. Only 10 messages across our experiments have a coverage less than 87%. This shows that highlighting only anomalies that explain 80% or more of their parent's latency deviation as root causes reduces false negatives and yet allows for a majority of root causes to be discovered in the first iteration.

7 Latency diagnosis with NSight

We now describe the iterative process by which NSight can quickly diagnose the causes of poor performance. Let us start with an initial system configuration that runs memcached in which large memcached tails are observed. Profiling the system with NSight identifies the prominent causes of tails. Users can mitigate or eliminate the causes found and profile the *reconfigured system* with NSight to identify the remaining causes of tails. Three such iterations help reduce memcached 99.9999th percentile latency from 15.3 ms to 182 μs in our setup. Unfortunately, some of the mitigations we use increase median latencies by a few microseconds. NSight helps identify *which* mitigations cause this increase by comparing profiling data collected *across* iterations.

We now describe these iterations with NSight and the notable causes of network latency in our system. We present diagnosis results only for memcached server *receive* latencies because the majority of end-host delays are known to show up on the receive path [35, 39, 78]).
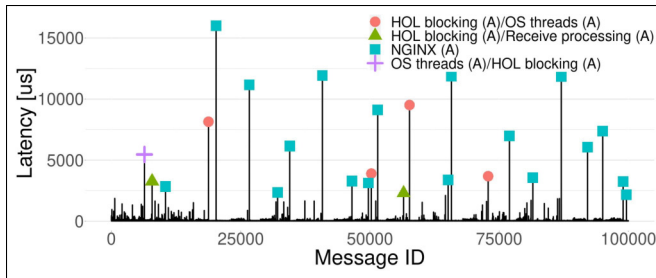
Figure 12: Balloon plot showing the top causes of tail latency with the initial configuration. The tallest balloons are blue squares, identified as *NGINX (A)* in the legend; this means `NGINX` interferes with `memcached` on the application core (A). The balloons are spaced apart, indicating that their causes are independent and can be addressed separately.

7.1 First iteration

Initial configuration. Our setup consists of two Linux machines (2x14 Intel Xeon Gold 5120, 192 GB RAM), running kernel version 4.18 (Ubuntu 18.04) connected by 100Gbps interfaces to a 3.2Tbps Ethernet switch. `memcached`, a latency sensitive workload (similar to [20, 33, 55]) runs alongside `NGINX`, the interfering workload. Both applications use the default configuration and share 4 cores. The workload consists of `memcached` requests arriving at 160-190k requests per second (60-80% of expected `memcached` throughput across 2 cores in our system), and `NGINX` requests arriving at 50-60k requests per second (60-80% of expected `NGINX` throughput on a single core). `ntp` and `ptp` were disabled in our setup when we collected the experimental results in this section.

Initial diagnosis. To use NSight, users can collect a profiling sample of a few seconds and look at the initial result. In our setup, this allows us to profile 100k `memcached` request-responses. The initial result, Figure 12, a balloon plot similar to Figure 3, shows the distribution of tail messages and their causes. The largest tails are caused by `NGINX` interference on the application core, *NGINX (A)*. The remaining tails are due to three causes, mainly head-of-line blocking of application threads, *HOL blocking (A)*, in combination with interference due to *OS threads (A)* and *Receive processing (A)* at the application core. These causes are defined in Table 1.

A second presentation, the box plot in Figure 13, summarizes the causes (X-axis) sorted by the *median* total deviation added to each tail message. The number of messages impacted by each cause, shown on the boxes, represents how *pervasive* the cause is. We notice that the most conspicuous causes impacting the most messages are to the right of *Receive processing (R)*, starting with *NGINX receive (A)* that impacts 596 tail messages. Going after these can increase the latency reduction achieved with this iteration.

Of the causes in Figure 13, we find that *HOL blocking (A)* impacts the most tail messages (950), indicating that there
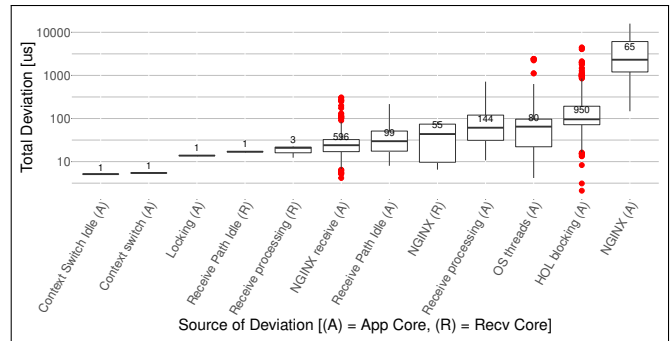
Figure 13: Global box plot showing the causes of tail latencies with the initial configuration. The number of tail messages impacted by each cause is shown on top of the boxes. This presentation helps identify the most important causes to pursue after one iteration of NSight profiling; pursuing both, causes that result in the largest deviations *and* causes that impact the *most tails*, can lead to the biggest latency reductions.

is a lack of I/O parallelism in `memcached`. When we examine the *tail* message lifetimes slowed down by *HOL blocking (A)* (the top outliers in the box plot), we see `memcached` *sends* delaying `memcached` request-*receives* by as much as 1 ms. To understand *why*, we look at `memcached` code and find that `memcached` threads process up to 20 requests per socket, sending responses for each, *before* processing the next request. This is interesting because papers [20, 55] have conjectured that it is sufficient to use microsecond granularity core-scheduling to improve `memcached` latencies, but in fact `memcached` itself foils that plan. The way to improve `memcached` latencies is to first modify `memcached` to introduce additional I/O parallelism as NSight identifies here.

7.2 Second Iteration

Second configuration. The next step is to mitigate or eliminate the causes identified in the previous diagnostic step. To eliminate *NGINX (A)* interference, we pin `memcached` to two cores and `NGINX` to a third core; similarly, to eliminate interference due to *Receive processing (A)* and *NGINX receive (A)*, we pin kernel receive activity to a fourth core and configure RSS to send all receive traffic to that core (Similar to IOKernel [55]). To mitigate *HOL blocking (A)*, we limit `memcached`-requests per socket to 2 (down from 20) and limit `memcached` server threads to one per core.

Second diagnosis. Having applied the second configuration we rerun NSight. Figure 14 and Figure 15 are the analogous presentations to Figure 12 and Figure 13. The largest latency deviations in Figure 15 come from *Paging/Paging Idle (A)*. The message lifetimes impacted by these causes show repeated calls to `change_protection` and `change_prot_numa` each taking up to 400 μs. The documentation for `change_prot_numa()` says that this code is a mechanism to identify beneficial page migrations; interestingly, it creates
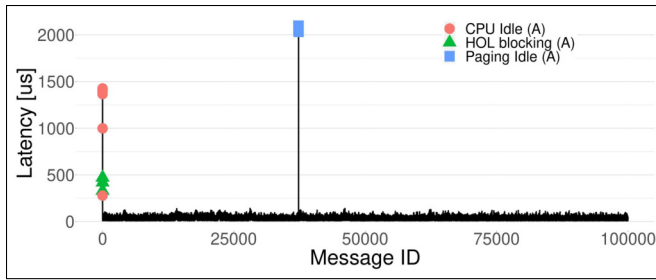
Figure 14: Balloon plot with the second configuration. Bunching of balloons shows system effects that impact message-bursts. Paging causes the largest tails. CPU idle activity and head-of-line blocking assail the initial few messages.
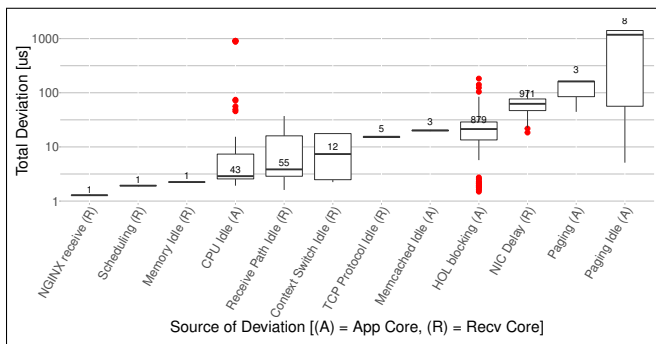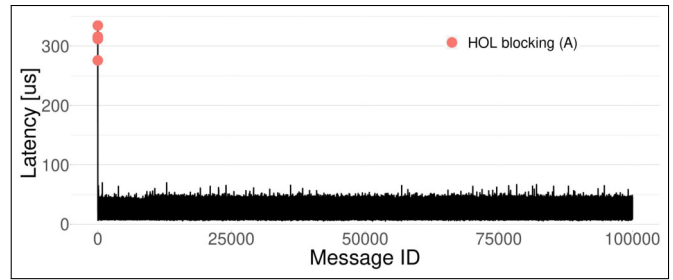

Figure 16: Balloon plot with the third configuration. A few balloons still appear, but their absolute latency (350 μs) is tiny compared to tails seen with the initial configuration (16 ms), and no balloons occur once the system is warmed up.

Figure 15: Global box plot with the second configuration. Alongside Paging and CPU idling that cause a few tails, this graph shows that interrupt coalescing (*NIC delay (R)*) and head-of-line blocking are most pervasive and worth pursuing.

deviant latencies. This suggests that architectural changes are needed to identify page migrations *without* causing tails. NSight can play a role by making it easy to confirm that the new architecture does not introduce latency deviations.

We also find that a few *CPU idle (A)* outliers in Figure 15 add up to 0.6 ms to tail messages; these correspond to the initial balloons in Figure 14. The lifetimes of these messages show large gaps in system activity *between* connection set up functions, dispatch_conn_new() called on the *receive core* to which we pinned the kernel receive activity, and conn_new() called on the memcached core. When we look at this code, we find that memcached registers an event handler for connection setup using libevent. When a new connection message is received (on the *receive core* in our setup) an event is dispatched to the handler on another thread that must be scheduled on the *application core*. The *delay in scheduling the connection handler* causes tail latencies during startup.

7.3 Third iteration

Third configuration. We now mitigate the causes found in Figure 15. We disable autonuma feature to confirm the *Paging/Paging Idle (A)* deviations, and adaptive interrupt coalesc-

ing to confirm *NIC Delay (R)* deviations, even though doing so might lead to worse performance. To speed up connection setup, we increase priority of new connection events by changing one line in memcached. Another cause we identified after the second iteration but before the third, was removed by disabling the LRU replacement feature; for compactness, we condense this additional iteration into the third iteration.

Third diagnosis. Having applied the third configuration we rerun NSight. The balloon plot Figure 16 shows that all the significant outliers are in the start up phase. The tail latencies of the initial messages are much lower than those seen with the second configuration in Figure 14 (350 μs vs. 1.4 ms). They are not caused by *CPU Idle (A)*, as they were with the second configuration, but by *HOL blocking (A)*. Overall, the tails are an order of magnitude smaller compared to the 1.1 ms-16 ms tails originally seen with the initial configuration in Figure 12.

7.4 Analysis of diagnosis and configurations

We now confirm tail latency improvements due to the diagnosis by running experiments with all three configurations, initial §7.1, second §7.2, and third configuration §7.3, for a longer duration without NSight profiling. We measure the latency of 10 million requests, using each configuration 10 times. Figure 17 shows the CDF of the receive latencies with these configurations; the left graph shows all the latencies and the right shows the tail latencies. With the third configuration, we see 99.9th percentile latency improve by 43× (2.2 ms to 51 μs). The 99.999th percentile latency is 67 μs and 99.9999th percentile latency is 182 μs (down from 15.3 ms). Both, the second and third configurations improve the tail latencies compared to the initial configuration but at the cost of *increasing* latencies in the first 60% of messages.

Diagnosing latency increases in lower percentiles. NSight analyses latency anywhere on a latency curve, not just the tail. Since the deviations in Figure 17 are worst around the 25th percentile Ⓒ (13.3 μs with initial but 24.7 μs with the third configuration), users can configure NSight to focus on the 25th percentile. NSight will analyse a slice of
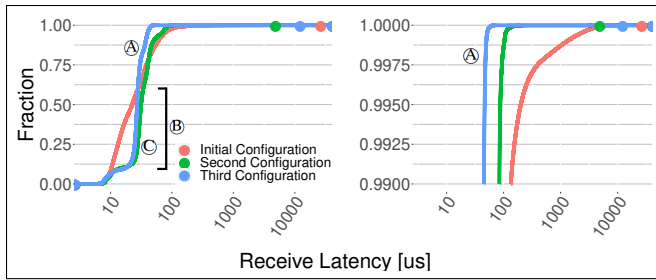
Figure 17: Full CDF (left) and p99-100 (right) of `memcached` receive latencies in longer runs (10 million requests, 10 repetitions) for all 3 configurations. The third configuration improves tail latencies beyond the 60th percentile Ⓐ but *adds* small latencies to messages between the 10th and 60th percentiles Ⓑ, with the maximum latencies being added *around* the 25th percentile Ⓒ; a proper solution instead of our expeditious mitigations could get the benefit of tail reduction without penalizing the common case.
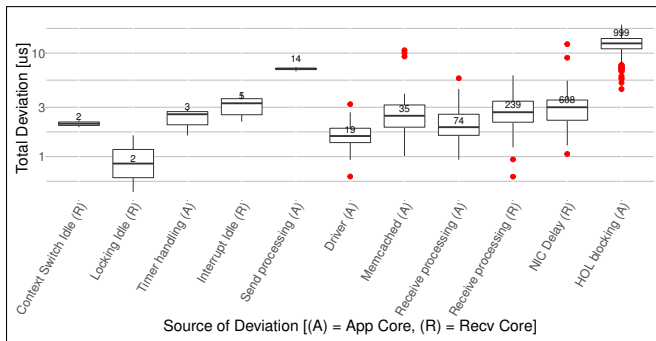
Figure 18: Global box plot comparing 25th percentile messages in the third iteration with those in the first iteration. To focus on causes that impact the *span* of messages around 25th percentile Ⓒ in Figure 16, the boxes are sorted by *number of messages impacted*. Head-of-line blocking is the most pervasive cause of the latency increase.

messages between the 20th and 30th percentiles. Since the latency increases occur *across iterations*, users can compare message lifetimes across iterations with NSight to identify the causes. We configure NSight to compare the 25th percentile messages in the third §7.3 and first iterations §7.1 to diagnose latency increases; the results in the rest of the section use this configuration. In this setting, latency deviation is defined as latencies that get worse around the 25th percentile in the third iteration compared to those in the first iteration.

Since *all* the messages around the 25th percentile are similarly impacted, we look for the *most pervasive causes*. Therefore, we turn to a global box plot that presents the causes *sorted by the number of messages impacted*, Figure 18. The most pervasive cause of latency, impacting all but one of the thousand messages around the 25th percentile in the third it-
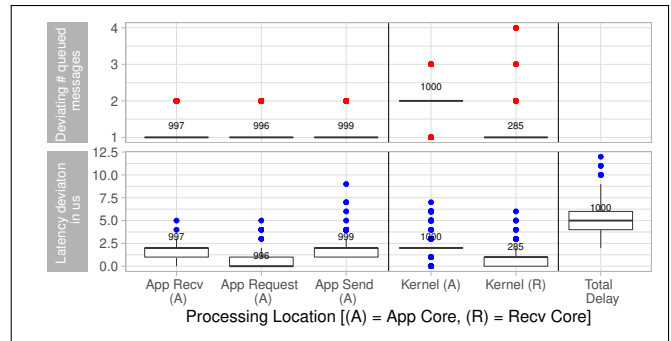
Figure 19: Queuing breakdown showing relative queuing ahead of 25th percentile messages in the third versus the first iteration. Deviations are shown for select functions (X-axis) for which we can measure the queue depth ahead of each message, by counting function occurrences in message lifetimes. The top half shows deviations in *queue depth*; a dot or line at 4 means that there are *4 more messages ahead of the deviant message relative to the reference*. The bottom half is a box plot for resulting latency deviations. The functions are categorized into application vs. kernel, send vs. receive vs. request processing. The count of messages impacted is shown on top of each plot. This graph shows that the 25th percentile messages experience more queuing in the third iteration.

eration relative to the first iteration, is *HOL blocking (A)*. The messages also experience *NIC delay (R)* relative to messages in the first iteration. This is surprising because we disabled adaptive interrupt coalescing in the third iteration §7.3.

To identify *why* the 25th percentile messages of the third iteration experience head-of-line blocking relative to the first iteration, we consult NSight's queuing breakdown described in Figure 19. It shows that a majority of the 25th percentile messages have at least one more message ahead of them in the third iteration relative to the first iteration (top half) and this introduces latency deviations (bottom half). *App Send (A)* shows more latency deviation than other categories. *Kernel (A)* and *Kernel (R)* measure send/receive queuing in the kernel. Because the second configuration (§7.2) isolated kernel receive activity from the *application* core, *Kernel (A)* shows latency deviations only due to *sending* responses. Receive queuing shown in *Kernel (R)* impacts fewer messages.

Together, deviations in *App Send (A)* and *Kernel (A)* show that *sending responses takes longer* in the third iteration even though *fewer responses* (reduced to 2 per socket in §7.2) are sent back per socket relative to the first iteration. This shows that the application core cannot keep up in the third iteration.

To find *why*, we consult NSight's core context summaries for the iterations, shown in Figure 20. It shows the percentage of time the *message lifetimes* spend in each processing context. We see that `memcached` uses three cores to process the 25th percentile messages in the first iteration, Figure 20a, compared to two in third iteration, Figure 20b. This confirms that the
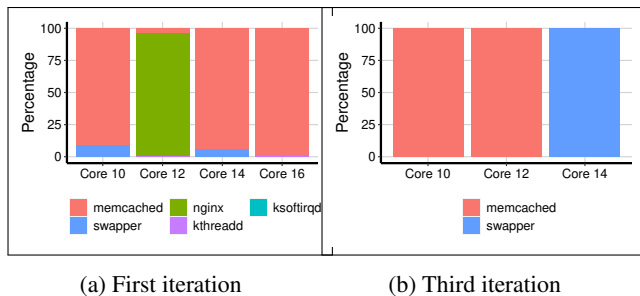
(a) First iteration   (b) Third iteration

Figure 20: NSight core-context summary showing the process contexts (legend) active in the 25th percentile `memcached` *message lifetimes* on each core (X-axis), and the percentage of time they are active (Y-axis). `memcached` has three cores available in the first iteration vs. two cores in the third iteration, exacerbating head-of-line blocking in the third iteration.

threads are backlogged in the third iteration because of the *lack of available CPU*; the mitigation of *pinning* `memcached` threads to cores (§7.1) increased head-of-line blocking for the 25th percentile messages in the third iteration.

Figure 20b also shows that in the third iteration, the core used for kernel receive activity is mostly idle/sleeping (`swapper` is the default idle task) waiting for a NIC interrupt, whereas in the first iteration the cores are always running, processing `memcached` requests or receiving messages in the kernel. This explains why we see deviations due to *NIC Delay (R)* in Figure 18 despite disabling adaptive interrupt coalescing; *waking the core* takes time and delays the NIC interrupt. This is an example of complex NIC–CPU scheduling interactions that NSight captures (§1). Waiting for the NIC interrupt at low loads puts the receive core to sleep *since it has nothing else to do*, and waking it up delays the NIC interrupt! Thus, the mitigation of pinning the kernel receive activity to a core in §7.2 adds latencies at lower loads to the common case.

8 Diagnosing VMA network stack

NSight can diagnose problems in different applications such as `redis` and different network stacks such as VMA network stack. We now describe the key causes of network tail latencies we found with unmodified `memcached` and `redis` on top of the VMA user-space network stack.

System configuration. For experiments with `memcached`, we pin `memcached` servers to 4 cores and increase the load (400k request/s on 4 cores). For experiments with `redis`, a single-threaded server, we profile a single core `redis` instance using the standard `redis-benchmark` (110k request/s on 1 core). Since these applications are already pinned to cores, studying application interference with `NGINX` is irrelevant and we do not include it in contrast to the Linux study.

Diagnosis. We now describe the key sources of network latency in VMA found using the same diagnostic strategy

we used in the Linux study, guided by NSight's graphs. As expected, the overall tail latency distribution improves with VMA in comparison to Linux. But surprisingly, the outliers for `memcached` are more severe. While the median `memcached` server request receive latency is 8.38 µs and the 99.9th percentile latency is 45.3 µs, the worst message latency is 34.5 ms! The median, 99.9th and worst `redis` request receive latencies are 1.63 µs. 145.1 µs, and 1.2 ms. Following are the most prominent causes of tails in VMA.

VMA epoll mechanism. NSight diagnostic graphs show that some `memcached` messages are delayed for up to 25 ms *in the NIC*! In this case, even though the message is received by the NIC, the epoll implementation of VMA does not pick it up. The exact reason for this behavior is unclear, but we posit that either the NIC hardware delays reporting the arrival or the stack waits for a message for a specific socket.

OS interference. We find that the default Linux scheduling policies frequently puts polling based stacks to sleep for short amounts of time (4 µs). In the case of some `memcached` messages that are severely delayed, we detect a kernel stack overflow, that puts all but one of the application threads to sleep for up to 12 ms with the lone thread handling the panic. This is unexpected behavior that seems to suggest a bug.

Buffer management. VMA ring buffer management causes frequent shorter latency deviations. It fills up the RX buffer queue with unused buffers and removes used send-buffers from the TX queue in bursts, adding deviations of up to 2.8 ms (though more frequently in the range of 60-150 µs).

9 Limitations and future work

We have already noted three limitations of NSight. First, NSight cannot be used for continuous profiling due to Intel-PT buffering implementation (§5.1). Second, CPU profilers capture system activity in batches of CPU cycles; NSight cannot capture latency deviations smaller than a batch (§5.2). Finally, consuming a core for §4.1, **Time reconciliation** is unnecessary if software clock changes are tracked by `ptp`. Another limitation of NSight is that it produces 600MB-1GB of compressed raw profiling data per second. Decompressing the profiling data to a usable format increases the size by 10-20x. Reducing the amount of data produced and speeding up analysis will reduce the time between NSight iterations.

We are expanding NSight's scope to RDMA-based stacks, and more general purpose performance diagnosis to analyze Linux's core operations [63]. We are also using NSight to characterize the design space for low-latency network stacks. We plan to make the tool available as open source.

Acknowledgements

References

[1] ebpf. Onlin., https://ebpf.io/.

[2] F-stack: High performance network framework based on dpdk. http://f-stack.org/.

[3] Ftrace. Onlin., https://www.kernel.org/doc/Documentation/trace/ftrace.txt.

[4] Perf: Performance analysis tools for linux. Onlin., http://man7.org/linux/man-pages/man1/perf.1.html.

[5] Seastar: High-performance server-side application framework. http://seastar.io/.

[6] Mohammad Mejbah ul Alam, Tongping Liu, Guangming Zeng, and Abdullah Muzahid. Syncperf: Categorizing, detecting, and diagnosing synchronization performance bugs. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 298–313, New York, NY, USA, 2017. Association for Computing Machinery.

[7] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using magpie for request extraction and workload modelling. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation - Volume 6*, OSDI'04, page 18, USA, 2004. USENIX Association.

[8] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, Broomfield, CO, October 2014. USENIX Association.

[9] Zachary Benavides, Keval Vora, and Rajiv Gupta. Dprof: Distributed profiler with strong guarantees. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

[10] M.Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: problem determination in large, dynamic internet services. In *Proceedings International Conference on Dependable Systems and Networks*, pages 595–604, 2002.

[11] Shuang Chen, Christina Delimitrou, and José F. Martínez. Parties: Qos-aware resource partitioning for multiple interactive services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 107–120, New York, NY, USA, 2019. Association for Computing Machinery.

[12] Intel Cooperation. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3B*. Intel Corporation, August 2007.

[13] Mathieu Desnoyers and Michel Dagenais. The lttng tracer: A low impact performance and behavior monitor for gnu/linux. *OLS (Ottawa Linux Symposium)*, 01 2006.

[14] The Kernel development community. Timestamping. Onlin., https://www.kernel.org/doc/html/latest/networking/timestamping.html.

[15] Jaana Dogan. Want to debug latency? https://rakyll.medium.com/want-to-debug-latency-7aa48ecbe8f7.

[16] Benjamin Donie. iostat. Onlin., http://man7.org/linux/man-pages/man1/iostat.1.html.

[17] Srikar Dronamraju. Uprobe-tracer: Uprobe-based event tracing. Onlin., https://www.kernel.org/doc/Documentation/trace/uprobetracer.txt.

[18] Úlfar Erlingsson, Marcus Peinado, Simon Peter, Mihai Budiu, and Gloria Mainar-Ruiz. Fay: Extensible distributed tracing from kernels to clusters. *ACM Trans. Comput. Syst.*, 30(4), November 2012.

[19] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation*, NSDI'07, page 20, USA, 2007. USENIX Association.

[20] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297. USENIX Association, November 2020.

[21] Junzhi Gong, Yuliang Li, Bilal Anwer, Aman Shaikh, and Minlan Yu. Microscope: Queue-based performance diagnosis for network functions. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 390–403, New York, NY, USA, 2020. Association for Computing Machinery.

[22] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. *SIGPLAN Not.*, 17(6):120–126, June 1982.

[23] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. Megapipe: A new programming interface for scalable network i/o. In *Proceedings of*

the 10th USENIX Conference on Operating Systems Design and Implementation, OSDI'12, page 135–148, USA, 2012. USENIX Association.

[24] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Nick McKeown, and Changhoon Kim. The nanopu: Redesigning the cpu-network interface to minimize rpc tail latency, 2020.

[25] Intel. Intel vtune profilder. Onlin., https://software.intel.com/en-us/vtune.

[26] Intel Corporation. Data plane development kit. https://www.dpdk.org/. April 2021.

[27] Calin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, and Junhua Wang. Perfiso: Performance isolation for commercial latency-sensitive services. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 519–532, Boston, MA, July 2018. USENIX Association.

[28] Alexey Ivanov. Optimizing web servers for high throughput and low latency. https://dropbox.tech/infrastructure/optimizing-web-servers-for-high-throughput-and-low-latency.

[29] Alan D. Brunelle Jens Axboe and Nathan Scott. blktrace. Onlin., http://man7.org/linux/man-pages/man8/blktrace.8.html.

[30] Masami Hiramatsu Jim Keniston, Prasanna S Panchamukhi. Kernel probes. Onlin., https://www.kernel.org/doc/Documentation/kprobes.txt.

[31] Nikolai Joukov, Avishay Traeger, Rakesh Iyer, Charles P. Wright, and Erez Zadok. Operating system profiling via latency analysis. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, page 89–102, USA, 2006. USENIX Association.

[32] Theo Julienne. Debugging network stalls on kubernetes. https://github.blog/2019-11-21-debugging-network-stalls-on-kubernetes/.

[33] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for microsecond-scale tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 345–360, Boston, MA, February 2019. USENIX Association.

[34] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter rpcs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 1–16, Boston, MA, February 2019. USENIX Association.

[35] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Amin Vahdat. Chronos: Predictable low latency for data center applications. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, New York, NY, USA, 2012. Association for Computing Machinery.

[36] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. Tas: Tcp acceleration as an os service. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[37] Chung Hwan Kim, Junghwan Rhee, Hui Zhang, Nipun Arora, Guofei Jiang, Xiangyu Zhang, and Dongyan Xu. Introperf: Transparent context-sensitive multi-layer performance inference using system stack traces. *SIGMETRICS Perform. Eval. Rev.*, 42(1):235–247, June 2014.

[38] John Levon. Oprofile. Onlin., https://oprofile.sourceforge.io/news/.

[39] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, page 1–14, New York, NY, USA, 2014. Association for Computing Machinery.

[40] ARM Limited. *ARM CoreSight Architecture Specification v3.0*. Intel Corporation, August 2017.

[41] Xiaofeng Lin, Yu Chen, Xiaodong Li, Junjie Mao, Jiaquan He, Wei Xu, and Yuanchun Shi. Scalable kernel tcp design and implementation for short-lived connections. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 339–352, New York, NY, USA, 2016. Association for Computing Machinery.

[42] Inc. Linux Kernel Organization. Linux. https://www.kernel.org/.

[43] Dan Luu. Sampling v. tracing. https://danluu.com/perf-tracing/.

[44] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, Denver, CO, June 2016. USENIX Association.

[45] Marek Majkowski. The story of one latency spike. https://blog.cloudflare.com/the-story-of-one-latency-spike/.

[46] Ilias Marinos, Robert N.M. Watson, and Mark Handley. Network stack specialization for performance. *SIGCOMM Comput. Commun. Rev.*, 44(4):175–186, August 2014.

[47] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: A microkernel approach to host networking. SOSP '19, page 399–413, New York, NY, USA, 2019. Association for Computing Machinery.

[48] Mellanox. Messaging accelerator (vma). Onlin., https://docs.mellanox.com/display/VMAv902.

[49] Microsoft. Event tracing for windows. Onlin., https://docs.microsoft.com/de-de/windows/win32/etw/about-event-tracing.

[50] Microsoft. Perfmon: Performance monitor on windows. Onlin., https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/perfmon.

[51] Sun Microsystems. Dtrace. Onlin., http://dtrace.org.

[52] David Miller. ethtool. Onlin., https://man7.org/linux/man-pages/man8/ethtool.8.html.

[53] MIPS. Pdtrace. Onlin., https://www.mips.com/develop/tools/navigator-probes/, August 2021.

[54] OpenZipkin. Zipkin: A distributed tracing system. Onlin., https://zipkin.io/.

[55] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, Boston, MA, February 2019. USENIX Association.

[56] P4. In-band network telemetry. Onlin., https://p4.org/p4-spec/docs/INT_v2_1.pdf.

[57] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T. Morris. Improving network connection locality on multicore systems. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, page 337–350, New York, NY, USA, 2012. Association for Computing Machinery.

[58] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16, Broomfield, CO, October 2014. USENIX Association.

[59] V. Prasad, William Cohen, F. Eigler, M. Hunt, J. Keniston, and B. Chen. Locating system problems using dynamic instrumentation. 01 2005.

[60] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 325–341, New York, NY, USA, 2017. Association for Computing Machinery.

[61] Linux PTP. The linux ptp project. Onlin., http://linuxptp.sourceforge.net/.

[62] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. Arachne: Core-aware thread management. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 145–160, Carlsbad, CA, October 2018. USENIX Association.

[63] Xiang (Jenny) Ren, Kirk Rodrigues, Luyuan Chen, Camilo Vega, Michael Stumm, and Ding Yuan. An analysis of performance evolution of linux's core operations. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 554–569, New York, NY, USA, 2019. Association for Computing Machinery.

[64] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. Pip: Detecting the unexpected in distributed systems. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, page 9, USA, 2006. USENIX Association.

[65] Luigi Rizzo. netmap: A novel framework for fast packet i/o. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 101–112, Boston, MA, June 2012. USENIX Association.

[66] Steven Rostedt. Kernelshark. Onlin., https://kernelshark.org/.

[67] Muhammad Shahbaz, Sean Choi, Ben Pfaff, Changhoon Kim, Nick Feamster, Nick McKeown, and Jennifer Rexford. Pisces: A programmable, protocol-independent software switch. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 525–538, New York, NY, USA, 2016. Association for Computing Machinery.

[68] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[69] Livio Soares and Michael Stumm. Flexsc: Flexible system call scheduling with exception-less system calls. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, Vancouver, BC, October 2010. USENIX Association.

[70] Brent Stephens, Aditya Akella, and Michael Swift. Loom: Flexible and efficient NIC packet scheduling. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 33–46, Boston, MA, February 2019. USENIX Association.

[71] Brent Stephens, Arjun Singhvi, Aditya Akella, and Michael Swift. Titan: Fair packet scheduling for commodity multiqueue NICs. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 431–444, Santa Clara, CA, July 2017. USENIX Association.

[72] Uber Technologies. Jaeger: open source, end-to-end distributed tracing. Onlin., https://www.jaegertracing.io/.

[73] VMware. Vprobes. Onlin., https://www.vmware.com/products/beta/ws/vprobes_reference.pdf.

[74] Kit Po Wong, Chi Ping Tsang, and Wan Yee Chan. Sherlock—a system for diagnosing power distribution ring network faults. In *Proceedings of the 1st International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems - Volume 1*, IEA/AIE '88, page 109–115, New York, NY, USA, 1988. Association for Computing Machinery.

[75] Wenfei Wu, Keqiang He, and Aditya Akella. Perfsight: Performance diagnosis for software dataplanes. In *Proceedings of the 2015 Internet Measurement Conference*, IMC '15, page 409–421, New York, NY, USA, 2015. Association for Computing Machinery.

[76] Minlan Yu, Albert Greenberg, Dave Maltz, Jennifer Rexford, Lihua Yuan, Srikanth Kandula, and Changhoon Kim. Profiling network performance for multi-tier data center applications. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, page 57–70, USA, 2011. USENIX Association.

[77] Fang Zhou, Yifan Gan, Sixiang Ma, and Yang Wang. wperf: Generic off-cpu analysis to identify bottleneck waiting events. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 527–543, Carlsbad, CA, October 2018. USENIX Association.

[78] Noa Zilberman, Matthew Grosvenor, Diana Andreea Popescu, Neelakandan Manihatty-Bojan, Gianni Antichi, Marcin Wójcik, and Andrew W. Moore. Where has my time gone? In *Passive and Active Measurement*, pages 201–214, Cham, 2017. Springer International Publishing.

CurvingLoRa to Boost LoRa Network Throughput via Concurrent Transmission

Chenning Li
Michigan State University

Xiuzhen Guo
Tsinghua University

Longfei Shangguan
University of Pittsburgh & Microsoft

Zhichao Cao
Michigan State University

Kyle Jamieson
Princeton University

Abstract

LoRaWAN has emerged as an appealing technology to connect IoT devices but it functions without explicit coordination among transmitters, which can lead to many packet collisions as the network scales. State-of-the-art work proposes various approaches to deal with these collisions, but most functions only in high signal-to-interference ratio (SIR) conditions and thus does not scale to real scenarios where weak receptions are easily buried by stronger receptions from nearby transmitters. In this paper, we take a fresh look at LoRa's physical layer, revealing that its underlying linear chirp modulation fundamentally limits the capacity and scalability of concurrent LoRa transmissions. We show that by replacing linear chirps with their non-linear counterparts, we can boost the throughput of concurrent LoRa transmissions and empower the LoRa receiver to successfully receive weak transmissions in the presence of strong colliding signals. Such a non-linear chirp design further enables the receiver to demodulate fully aligned collision symbols — a case where none of the existing approaches can deal with. We implement these ideas in a holistic LoRaWAN stack based on the USRP N210 software-defined radio platform. Our head-to-head comparison with two state-of-the-art research systems and a standard LoRaWAN baseline demonstrates that CurvingLoRa[1] improves the network throughput by 1.6–7.6× while simultaneously sacrificing neither power efficiency nor noise resilience.

1 Introduction

As we gradually reach a cyber-physical world where everything near and far is connected wirelessly, a fundamental question worth discussing is which wireless technologies are best suited for achieving this goal. While Wi-Fi and cellular networks have proved their success in provisioning high-throughput wireless connectivity in small geographic areas, a remaining challenge is connecting those low-power IoT devices deployed in wide areas. Most of these devices are powered by batteries and thus require minimal communication overhead.

Long Range (LoRa) [2], SIGFOX [7], and NB-IoT [38] are the three commercialized wireless technologies facilitating low-power wide-area IoT deployments. LoRa is an open-source technique operating at the unlicensed ISM Sub-GHz bands, without subscription fees [26]. Central to LoRa is a dedicated PHY-layer design that leverages Chirp Spread Spectrum (CSS) modulation to facilitate packet decoding in extremely harsh signal-to-noise ratio (SNR) conditions (which can be as low as -20 dB [58]). Coupling with a long-term duty cycling mechanism, a deployed LoRa node can last for a few years with a single dry-cell battery. These dual merits of low-power and long-range make LoRaWAN an attractive solution for IoT connectivity outdoors.

Unlike Wi-Fi, which uses *carrier sensing* [6] to avoid packet collisions, LoRa's communication protocol LoRaWAN functions without explicit coordination due to its stringent power budget. It instead adopts the least restrictive MAC protocol—ALOHA [1]—that allows participating nodes to transmit immediately once they wake up.[2] Such *laissez-faire* transmission inevitably causes packet collision when multiple LoRa nodes transmit simultaneously, resulting in retransmissions that can drain the battery of collision nodes and crowd the precious wireless spectrum on the unlicensed band [13]. Packet collisions are exacerbated with increased network size, thus reducing throughput and fundamentally challenging LoRa networks' scalability in real deployment [15]. For example, the probability of packet collisions grows from 1% to 10% when the LoRaWAN network size scales from 100 to 1000 nodes [40], thus restricting many large-scale applications such as factory automation [17, 34], smart city [7, 30], data-driven agriculture [37, 43], and smart metering [9, 50].

In this paper, we take a fresh look at the physical layer design of LoRaWAN and reveal that the underlying *linear chirp*

---

[1]Code is available at *https://github.com/liecn/CurvingLoRa_NSDI22*

[2]LoRaWAN recently released a new feature called Channel Activity Detection (CAD) that allows the receiver to scan the channel before transmitting. However, CAD incurs extra power consumption and thus may not apply to rural deployments where battery replacement is usually infeasible.

based modulation fundamentally limits the capacity and scalability of concurrent LoRa transmissions. We present CurvingLoRa, a simple but effective PHY-layer design to boost the LoRa network throughput by simply replacing the standard linear chirp with its nonlinear chirp counterpart.

CurvingLoRa is based on a unique property of non-linear chirps, which we term the *energy scattering and converging effect*. When a non-linear up-chirp symbol misaligns with the non-linear down-chirp during demodulation, their multiplication will *spread* the power of the non-linear up-chirp symbol into multiple FFT bins where the associated energy peaks are inherently weak. Such energy scattering effect will show up as long as the non-linear up-chirp is not well aligned with the down-chirp (The theoretical analysis is in Appendix A). In contrast, when this non-linear up-chirp is well aligned with the down-chirp, its signal power will *converge* to a specific frequency point, leading to a strong energy peak after FFT, as shown in Figure 1.

This property allows the receiver to manipulate the signal-to-interference ratio (SIR[3]) of each collision symbol for reliable demodulation. In contrast, the power of linear chirps will always be *converging* to a single frequency point regardless of its alignment with the down-chirp in the demodulation process. This energy converging effect fundamentally limits the decodability of linear chirps in the presence of collisions.

We analyze the performance of non-linear chirp and compare it with its linear chirp counterpart in various SNRs, SIRs, and symbol overlapping ratio conditions. We show that such a non-linear chirp remarkably improves the transmission concurrency while retaining high power efficiency and strong noise resilience as linear chirp does (§4). We then design a holistic PHY layer to realize non-linear chirp modulation and demodulation (§5) and implement it on software-defined radios for evaluation. The experimental results show that compared to two state-of-the-art systems [47, 53], CurvingLoRa can effectively improve network throughput by $1.6 - 6.6\times$ and $2.8 - 7.6\times$ in an indoor and outdoor deployment. In addition, we make the following contributions:

- We reveal that LoRaWAN's PHY-layer design fundamentally limits the transmission concurrency and propose a simple but effective solution. CurvingLoRa takes advantage of the power scattering effect of non-linear chirps to enable LoRa concurrent transmissions in extreme SNR, SIR, and symbol overlapping ratio conditions.

- Through theoretical analysis and experimental validation, we demonstrate that CurvingLoRa outperforms both the current practice and the standard LoRaWAN without sacrificing the power efficiency, noise resilience, or data rates. These desired properties make non-linear chirp a potential complement to its linear chirp counterpart.

---

[3]Defined as the ratio between the power of the targeting LoRa chirp and the power of interfering concurrent LoRa chirps.
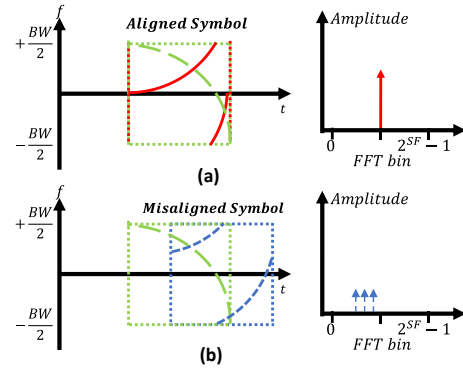

Figure 1: An illustration of CurvingLoRa's energy converging and scattering effect. (a): The energy of a non-linear chirp symbol will converge to a specific frequency point when it aligns with the down-chirp. (b): The energy will spread into multiple FFT bins when this non-linear chirp mismatches with the down-chirp.

- We design a holistic PHY-layer and implement it on a software-defined radios platform to evaluate CurvingLoRa in various real-world scenarios. The results confirm that the CurvingLoRa can greatly improve the network throughput.

2 Related Work

Resolve collisions at PHY layer. Prior works on resolving LoRa collisions have followed a common theme: exploring the unique features of collided LoRa symbols in the time domain [22, 53, 55, 59], frequency domain [13, 42, 47, 56], or both [20, 41]. For instance, mLoRa [53] observes that collisions usually start with a stretch of interference-free bits on the packet header. The receiver can thus decode these uncontaminated bits first and then leverage successive interference cancellation [16, 33] to decode the collided bits iteratively. Choir [13] uses the frequency variation caused by oscillator imperfection to map bits to each LoRa transmitter. FTrack [55] jointly exploits the distinct tracks on the frequency domain and misaligned symbol edges in the time domain to separate collisions. By combining spectra obtained from different parts within each symbol, CIC [41] exploits the sub-symbols that provide both time and frequency resolution to cancel out the interference under low SNR conditions.

While the above ideas have demonstrated their efficacy, they still face two scalability issues that fundamentally challenge their applicability in practice: First, the vast majority of these approaches do not scale to many concurrent transmissions. For instance, mLoRa [53] and FTrack [55] barely support up to three concurrent transmissions to maintain a symbol error rate less than 0.1. While Choir [13] improved over the above methods, it does not scale to more than ten devices. Although NScale [47] can support tens of concurrent transmissions, it requires the overlap ratio between different symbols to be lower than a rigid threshold, which is unlikely

to be held in practice given laissez-faire LoRa transmissions. Second, none of the foregoing approaches scale well to near-far deployment scenarios. Since after dechirping, the weak reception from a remote transmitter produces a tiny FFT peak that is likely to be buried by strong FFT peaks from LoRa nodes that are closer to the receiver.

Although successive interference cancellation (SIC) can be leveraged to deal with this near-far issue, it functions only in high SNR conditions due to the following reasons. First, due to Carrier Frequency Offset (CFO) and Sampling Frequency Offset (SFO), the phase of received chirp symbols is likely to distort by a certain degree. This phase distortion is critical to the signal recovery in SIC but is difficult to estimate in low SNR conditions [42]. Second, SIC suffers from hardware imperfections [47], which is common on low-end IoT devices. As a result, the symbol recovering error accumulates gradually and is likely to fail the SIC in the end. In addition, the impact of ambient RF noise on SIC, particularly the parameter estimation for signal reconstruction, gets exacerbated at low SNR conditions. Instead of leveraging new features on time or frequency domain to combat LoRa collisions, CurvingLoRa addresses this issue from a fresh new angle and designs new types of chirp symbols to facilitate concurrent transmissions.

Resolve collisions at other layers. Significant research efforts have been made to address signal collisions from the perspective of MAC-layer. For instance, by leveraging Channel Activity Detection (CAD) [15, 51] or deep neural networks [8], a plenty of works [8, 15, 51] propose carrier-sense multiple access (CSMA)-based MAC protocol to avoid LoRa collisions. There are also some works explore special coding mechanism and MAC-layer co-design [11, 18, 31, 40, 60] to alleviate or even avoid LoRa packet collisions. For example, NetScatter [18] presents a distributed CSS coding mechanism by assigning each IoT device a different chirp symbol. Multiple LoRa devices can then transmit concurrently through ON-OFF Keying modulation. Another way to alleviate the impact of collisions is adding data redundancy (e.g., convolutional codes, Viterbi decoder) to correct bit errors in corrupted frames at MAC layer. For example, DaRe [31] combines the convolutional and fountain codes for data recovery in the presence of a frame loss. CurvingLoRa can leverage such MAC-layer optimization and data recovery algorithms to further improve the system performance.

Non-linear Chirp for Communication and Radar. Non-linear frequency modulation has been widely used in radar systems. Lesnik *et al.* [25] demonstrate that using nonlinear frequency modulation can enhance signal sensitivity. Doerry *et al.* [12] and Benson *et al.* [4] detail the way to build non-linear chirp receivers. Kahn *et al.* [23] and Hosseini *et al.* [19] use nonlinear chirps in a Multi-user orthogonal chirp spread spectrum (MU-OCSS) communication system to mitigate the multiple access interference problem. Wang *et al.* [52] propose to use non-linear chirps for communication systems of
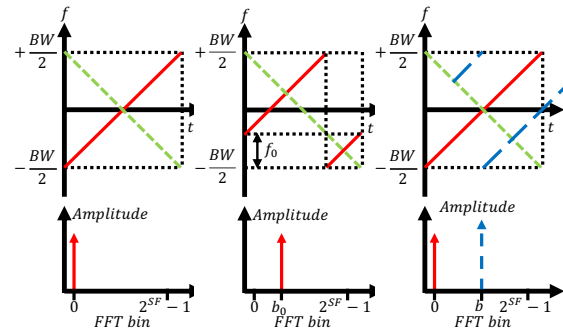
(a) Baseline up-chirp (b) Shifted up-chirp (c) Symbol collision

Figure 2: LoRa PHY-layer design. (a): the multiplication of an up-chirp and a down-chirp leads to an energy peak on a specific FFT. (b)This energy peak's position varies with the initial frequency offset of the incoming up-chirp. (c): Two collided symbols have separate energy peaks on FFT bins.

binary orthogonal keying mode. In contrast, CurvingLoRa explores a new possibility of using non-linear chirps to improve the reception of concurrent LoRa-like signals.

3 Motivation

This section briefly introduces the LoRa physical layer and then analyzes the pros and cons of linear chirps (§3.1). A discussion on the limitations of resolving linear-chirp LoRa collisions follows (§3.2).

LoRa Physical Layer. LoRa modulates data with chirp spread spectrum (CSS) [5, 13]. The transmitter encodes bits by varying the initial frequency offset of a standard up-chirp.[4] For instance, bits '00' are encoded by an up-chirp with zero initial frequency offset, while bits '01' are encoded by shifting the initial frequency by f_0. The frequency component beyond $BW/2$ will be wrapped to $-BW/2$, ensuring full bandwidth occupancy. The receiver (e.g., a LoRa gateway) first detects the incident LoRa packet through correlation (§5.2). To demodulate the packet, the receiver multiples each chirp symbol with a standard base down-chirp. The multiplication leads to an FFT peak in the frequency domain, which allows the receiver to demodulate LoRa chirp symbols by detecting the position of FFT peaks. Figure 2(a)-(b) shows an example.

3.1 The Pros and Cons of Linear Chirp

In essence, the aforementioned *dechirp* converges the power of each LoRa symbol to a specific frequency point (*i.e.*, an energy peak on an FFT bin), which allows the LoRa chirp to be decodable in extremely low SNR conditions (*i.e.*, -20 dB [58]). As more LoRa nodes get involved, we are expected to see packet collisions at the receiver since LoRa

---

[4]A chirp signal whose frequency grows linearly from $-BW/2$ to $BW/2$.
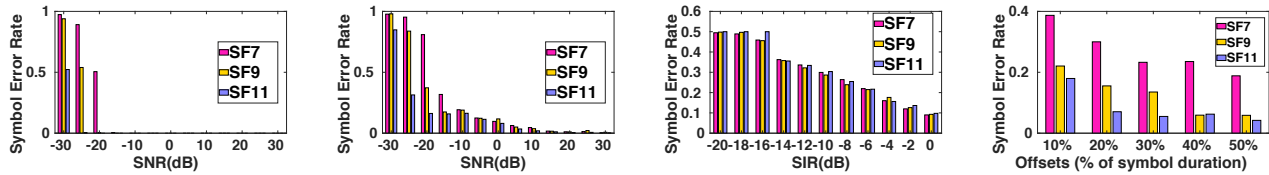
(a) SER fluctuates with SNR in the absence of collisions.

(b) SER fluctuates with SNR in the presence of collisions ($SIR \approx 0dB$).

(c) SER fluctuates with SIR in the presence of collisions ($SNR > 30dB$).

(d) SER fluctuates with symbol offset in the presence of collisions.

Figure 3: Evaluation of the successive interference cancellation-based collision resolving method [53] in various settings.

nodes abide by the least-restrictive MAC protocol ALOHA. To solve packet collisions, LoRaWAN [2] stipulates a set of spreading factors (SF) (*i.e.*, 7-12) and different bandwidths (BW) (*i.e.*, 125/250/500 KHz). Therefore, LoRa packets with different SFs or BWs can transmit concurrently on the same frequency band. The receiver uses down-chirps with different SFs to disambiguate these concurrent transmissions. However, the throughput of this regulation is limited: it supports only 18 pairs of SF&BW combinations [8, 18].

Collision happens when two concurrent transmissions use the same SF and BW. In this case, we are expected to see two energy peaks in two separate FFT bins, as shown in Figure 2(c). In practice, due to the near-far issue, one transmission (e.g., packet *A* in red) may experience a stronger attenuation than the other (e.g., packet *B* in blue). Hence the energy peak of *A* tends to be weaker than that of *B* in FFT bins. Accordingly, the receiver will only take *A* as noise and demodulate *B*. When *A* and *B* experience similar attenuation, the receiver can reliably demodulate neither of them because their individual energy peak may bury each other across different symbols. In a nutshell, when two LoRa packets collide, only the strongest transmission can be correctly demodulated by LoRaWAN.

3.2 Resolving Linear-Chirp LoRa Collisions

Section 2 overviews the current practice on resolving LoRa collisions and explains their pros and cons. This section implements a state-of-the-art SIC-based system, mLoRa [53], and examines its performance in various SNR and SIR conditions. We also compare it with other SOTA systems in the evaluation part. Specifically, we first measure the noise resilience of a standard LoRa packet in the absence of collisions (Figure 3(a)). We then synthesize symbol collisions and measure their symbol error rate (SER) in different SIR and SNR settings. To achieve this goal, we collect multiple pairs of LoRa packets and superimpose them together with a symbol offset varying from 0 to 50% of the symbol time. We then emulate different SIR and SNR conditions by adding Gaussian white noises and varying the amplitude of superposed packets, respectively. We finally measure the SER in different SNR and SIR conditions.

From Figure 3(a), we observe that the LoRa receiver can reliably decode a collision-free LoRa packet (i.e., SER<1%) even the SNR of this packet drops to -20dB [27, 58]. However, to maintain the same SER for a collision symbol, the SNR

of this collision symbol should be $5dB - 25dB$ higher than that of a collision-free LoRa symbol, as shown in Figure 3(b). Such a high SNR requirement sets a strong barrier for the practical adoption of mLoRa since LoRa transmissions tend to be very weak after attenuation over a long distance. We also observe that the SER grows dramatically with the decreasing SIR (Figure 3(c)), indicating that mLoRa [53] cannot reliably demodulate the weak targeting LoRa symbols (i.e., $SIR < 0dB$) in the presence of strong concurrent LoRa transmissions. Furthermore, we observe that the SER grows with decreasing symbol offset (Figure 3(d)), which confirms our analysis.

Remarks. The above analysis reveals that the linear chirp in LoRaWAN does not scale to concurrent LoRa transmissions. Although the state-of-the-arts have proposed various approaches to resolve LoRa collisions, most of them function only in good SNR or SIR conditions and thus sacrifice the precious processing gain brought by the chirp modulation.

4 Analysis: Non-linear vs. Linear Chirps

We now show that by replacing the linear chirp with its non-linear counterpart, we can boost the capacity of concurrent transmissions (§4.1) while allowing the receiver to demodulate collision signals in severe SIR conditions (§4.2). In addition, we show by both theoretical analysis and empirical validation that such a non-linear chirp design sacrifices neither noise resilience (§4.3) nor power efficiency (§4.4).

4.1 Non-linear Chirps Meet Collisions

We define a non-linear up-chirp as a signal whose frequency grows non-linearly from $-BW/2$ to $BW/2$. The non-linear function can be polynomial, logarithmic, exponential, or trigonometric. The receiver operates dechirp to demodulate non-linear chirp symbols.

Considering two collision symbols *A* and *B*, as shown in Figure 5(a). The receiver takes a sliding window approach to demodulate incoming signals. As aforementioned, when symbol *A* aligns with the down-chirp in the current observing window, we are expected to see a strong energy peak (termed as peak *A*) on the associated FFT bin. At the same time, the energy of symbol *B* will be spread over multiple, clustered FFT bins due to its misalignment with the down-chirp. Compared to peak *A*, the amplitude of these clustered energy peaks
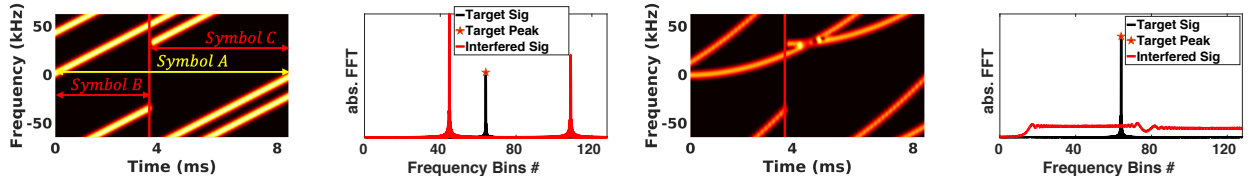
(a) L. chirp collisions in time domain (b) L. chirp collisions in freq. (c) NL. chirp collisions in time domain (d) NL. chirp collisions in freq.

Figure 4: Comparison of linear and non-linear chirps (i.e., $f(t) = t^2$) on resolving the near-far issue. (a): Three linear chirps with different SNR collide at the receiver. (b): Due to the near-far issue, the energy peak (in black) of the weak reception is overwhelmed by the energy peaks (in red) of strong collision symbols. (c): Three non-linear chirps in the same collision situations. (d): The spectral power of strong receptions is spread over multiple frequency points, making the corresponding FFT peaks (in red) significantly lower than the converged power (in black) of the weak reception.
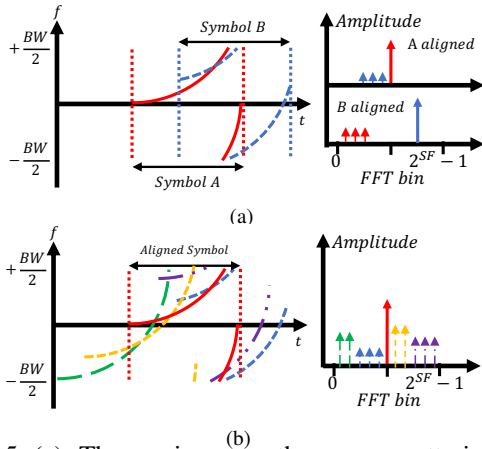
Figure 5: (a): The receiver uses the power scattering effect to demodulate two collision symbols. (b): an illustration of demodulating five collision symbols.

is significantly weaker. The receiver inherently takes these clustered FFT peaks as noise and demodulates symbol *A*. As the observing window slides, symbol *B* will align with the down-chirp at a time. Consequently, the dechirp converges the power of symbol *B* to a specific frequency point while spreading the power of symbol *A* into multiple frequency bins instead. The receiver then takes symbol *A* as noise and demodulates *B*. When it comes with two collision packets with each consisting of tens of symbols, following the same principle, the receiver slides the observing window step by step and demodulates their individual symbols alternatively.

Remarks. In essence, when the collision symbols are not strictly aligned, there will be only one symbol that aligns with the down-chirp in each observing window. This indicates that each time only one symbol gets its energy accumulated. In contrast, the energy of all the other colliding signals is being scattered over multiple FFT bins, as shown in Figure 5(b). Hence the receiver can easily pick up each symbol on separated observing windows and decode them chronologically.

4.2 Accounting for the Near-far Effect

The above section explains the basic idea of non-linear chirp and its unique energy scattering effect. We next demonstrate

that this energy scattering effect can be leveraged to address the *near-far issue* where weak receptions are buried by strong receptions from nearby transmitters.

Consider many colliding transmitters where some are physically closer to the receiver than the others. When linear chirps are adopted, the power of strong receptions *converges* to a specific frequency point where the associated energy peak is easily distinguishable after dechirp. However, the weak receptions from remote transmitters have significantly weaker energy peaks. The receiver thus takes those weak receptions as noise. Figure 4(a) shows a snapshot where two linear chirp packets with a distinguishable SIR (-10dB) collide at the receiver. Suppose the current observing window aligns with the symbol *A* in yellow of the weak packet. Due to the near-far issue, symbols *B* and *C* in red produce stronger energy peaks on associated FFT bins even they both are not aligned with the current observing window (shown in Figure 4(b)). Hence the receiver cannot demodulate symbol *A* successfully. In contrast, when non-linear chirps are adopted (shown in Figure 4(c)), the power of strong reception symbols *B* and *C* are both *scattered* into multiple FFT bins after dechirp. Due to such an energy scattering effect, the energy peaks induced by these strong symbols become lower than the accumulated energy peak induced by the weak symbol *A*. This allows the receiver to demodulate symbol *A* in the presence of strong collision symbols *B* and *C* (shown in Figure 4(d)).

Validation. To demonstrate the effectiveness of non-linear chirps on resolving the near-far issues, we compare the SER of non-linear (i.e., the quadratic function: $f(t) = t^2$) and linear chirps (i.e., $f(t) = t$) in different SIR settings. To create collision symbols, we collect the targeting and interfering LoRa packet seperately using a USRP N210 software defined radio. We then superimpose these two packets together on software. The SF, BW, and sampling rate are set to 10, 125KHz, and 1MHz, respectively.

Figure 7(a) shows the results. Per our analysis, we observe that the linear chirp fails to demodulate the targeting symbol in the presence of strong concurrent transmissions (i.e., SER=100% when SIR<−5dB). The SER then drops to around 10% when the power of the targeting symbol is comparable to that of colliding symbols (i.e., SIR=0dB). In
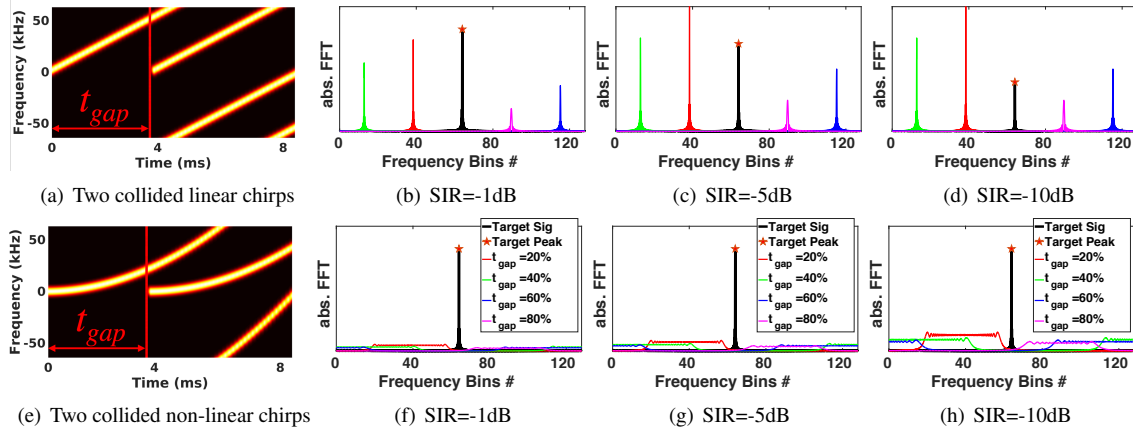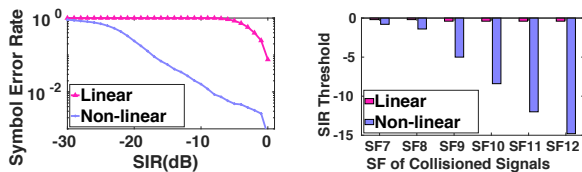
Figure 6: Examining the resilience of non-linear chirps (i.e., $f(t) = t^2$) to symbol time offset in various SIR settings.

(a) SER of linear and non-linear chirps in different SIR conditions.

(b) The SIR threshold for linear and non-linear chirps to achieve 1% SER.

Figure 7: Comparing the SER of linear and non-linear chirps in various near-far conditions.

contrast, the receiver can successfully demodulate the weak non-linear symbols reliably (i.e., $SER < 1\%$) as long as the SIR is higher than $-10dB$. We also found that the non-linear chirp can still achieve 10%+ SER in an extreme case where the colliding signal is $20dB$ stronger than the targeting signal (i.e., SIR=$-20dB$). We further evaluate the SER in different SF settings. Figure 7(b) shows the minimum SIR required by each type of chirps to achieve less than 1% symbol error rate. We observe that the linear chirp requires a minimal SIR of around $0dB$ in all six SF settings. In contrast, the non-linear chirps require a minimal SIR less than $0dB$, and the SIR requirement drops dramatically with increasing SF. These results clearly demonstrate that the non-linear chirp by its own design is more scalable to near-far issues than its linear chirp counterpart.

The impact of symbol time offset. We define t_{gap} as the symbol time offset between two colliding symbols A and B (shown in Figure 6(a)). Suppose the current observing window aligns with symbol A, then after dechirping, the power of the interfering symbol B will be scattered into multiple FFT bins. The amplitude of these scattered FFT peaks is proportional to 1-t_{gap} because only those signal samples that are within the overlapping window will contribute to the energy peaks. Hence a smaller t_{gap} will lead to stronger interfering peaks. We vary t_{gap} and plot the energy peaks in Figure 6.

Figure 6(b) shows the energy peaks of linear chirps. When SIR=$-1dB$, the targeting peak is still distinguishable from

the interfering peak across all four t_{gap} settings. When SIR drops to $-5dB$ (shown in Figure 6(c)), the interfering peak grows dramatically with decreasing t_{gap}. It finally surpasses the targeting peak when t_{gap} drops to 20%. When SIR grows to -10dB (shown in Figure 6(d)), the interfering peak easily exceeds the targeting peak in 3/4 t_{gap} settings. In contrast, when a non-linear chirp is adopted, we merely observe tiny energy peaks induced by the interfering symbol B. The targeting peak is easily distinguishable even in the case that the colliding symbol B is almost aligned with the targeting symbol A (i.e., t_{gap}=20%) across all three SIR settings, as shown in Figure 6(f)-(h). These results manifest that the non-linear chirp is robust to symbol time offset. In §5.1 we further demonstrate that by adopting different forms of non-linear chirps, the receiver can even demodulate two well-aligned collision symbols (i.e., t_{gap}=0) — a case that none of existing LoRa collision demodulation approaches can deal with.

4.3 Noise Tolerance

Theoretically, the noise tolerance of CSS symbol is determined by the symbol bandwidth and symbol time [27, 42, 47]. CurvingLoRa's non-linear chirps are of equal length to LoRa linear chirps and occupy the same bandwidth. Thus we expect the non-linear chirps can achieve the same noise tolerance as their non-linear linear chirp counterparts.

Validation. We evaluate the noise tolerance of six non-linear chirps that cover a range of shapes and convexity (§5.1):

(1): *quadratic1—$f(t) = t^2$*	(2): *quadratic2—$f(t) = -t^2 + 2t$*
(3): *quartic1—$f(t) = t^4$*	(4): *quartic2—$f(t) = -t^4 + 4t^3 - 6t^2 + 4t$*
(5): *Sine1—$f(t) = sin(t), t \in [-\pi/2, \pi/2)$*	(6): *Sine2—$f(t) = sin(t), t \in [-3\pi/8, 3\pi/8)$*

Figure 8(a) shows the SER achieved by these chirps in three SF settings. We observe that all these six types of non-linear chirps achieve consistent symbol error rates with their linear chirp counterpart across all three different SF settings. In particular, when $SF = 11$, the receiver achieves 1% SER on both non-linear and linear chirps in an extremely low SNR condition (i.e., $-20dB$). The minimal SNR (for achieving the same SER) then grows to $-14dB$, and further to $-9dB$ as the
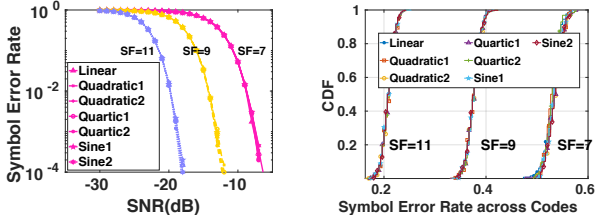
(a) SER fluctuates with SNR in vari- (b) The CDF of SER in various SF
ous SF settings. settings.

Figure 8: Comparing the SER of various types of non-linear chirps in different SNR conditions.

SF drops to 9 and 7, respectively. The result demonstrates that the non-linear chirp achieves superior noise tolerance as the linear chirp does.

Since the LoRa symbol varies with the chirp's initial frequency offset, given a certain type of non-linear chirp, one may worry that the SER of this chirp may not be consistent across different LoRa symbols. To validate this concern, we generate different chirp symbols by varying the initial frequency offset of a standard up-chirp. We then compare its symbol error rate with linear-chirps in the same SF settings. As shown in Figure 8(b), the linear and non-linear chirps achieve very similar SER in all three SF settings.

4.4 Power Consumption

Next, we show that the non-linear chirp generation consumes the same order of power as the linear chirp generation does. We leverage Direct Digital Synthesis (DDS) [49], a digital signal processing method to generate chirp signals. Compared to other analog frequency synthesis [39] or voltage-controlled oscillator (VCO) [46] based approaches, DDS is immune to both frequency and amplitude drifts and thus has been widely adopted for chirp signal generation in a radar system, e.g., frequency modulated continuous wave (FMCW) radars [29, 32] and synthetic aperture radars (SAR) [24, 57].

DDS works as follows. It first generates a reference signal at a constant frequency f_{clk}, and stores the signal samples in a local buffer, called a phase-amplitude mapping table. Let L be the length of this mapping table. To generate a desired chirp signal, DDS then accesses the mapping-table following the equation defined below:

$$\phi_i = \sum_{m=1}^{i} f_i = \sum_{m=1}^{i} (f_{i-1} + K_i \times \frac{f_{clk}}{2^L}) = \sum_{m=1}^{i} \sum_{j=1}^{m} K_i \times \frac{f_{clk}}{2^L} \quad (1)$$

where ϕ_i and f_i represent the phase and frequency of the ith sampling point of the chirp signal to be generated, respectively. K_i is the slope of this chirp signal, describing how its frequency changes over time. K_i is a constant value for linear chirps. It varies over time for non-linear chirps. The transmitter then retrieves these signal samples from the mapping table and generates the chirp signal accordingly.
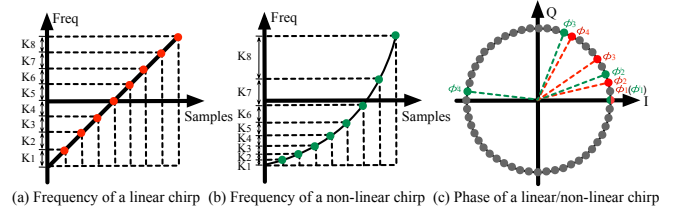

(a) Frequency of a linear chirp (b) Frequency of a non-linear chirp (c) Phase of a linear/non-linear chirp

Figure 9: An illustration of DDS operation to generate the linear and non-linear chirps, respectively.

Figure 9 describes DDS's high-level operations. To generate a linear chirp (Figure 9(a)), the transmitter sets K_i to a constant value (i.e., 1) and accesses the frequency samples at index (1, 2, 3, 4, ...). The phase samples are retrieved at index (1, 3, 6, 10, ...). In contrast, to produce a non-linear chirp (Figure 9(b)), K_i varies over time, e.g., (K_1=1, K_2=2, K_3=4, K_4=8, ...). The frequency and phase index then changes to (1, 3, 7, 15, ...) and (1, 4, 11, 26, ...), respectively.

Validation. We prototype DDS on a Zynq-7000 FPGA [36] and measure the power consumption of linear and non-linear chirp generation, respectively. The FPGA board is equipped with an ultra-low-power 12-bit ADC and a 256KB RAM. The phase-amplitude mapping table is generated by a 1 MHz clock signal. It stores 2^{12} sample points. We then retrieve signal samples from this mapping table to generate chirps (BW=125KHz, SF=7). The sampling points of each chirp in total are 8192. Our measurement study shows that the transmitter consumes the same order of power on generating the baseband of these two types of chirp signals: 315.6 μW for non-linear chirps and 306.2 μW for linear-chirps, respectively. The up-conversion of baseband to RF band (900 MHz) consumes around 40 mW [21] for both chirps. Hence the total power consumption (baseband+RF) of the DDS-based approach is similar to commercial LoRa nodes [10].

5 CurvingLoRa PHY-Layer

The above section shows a set of desirable properties of non-linear chirps. In this section, we describe the PHY-layer design on non-linear chirp modulation (§5.1), demodulation (§5.2), and the frame format for packet detection (§5.3).

5.1 Modulation

Similar to the standard linear chirp modulation in LoRa, CurvingLoRa defines a base non-linear chirp and modulates it by varying its initial frequency offset.

Base non-linear chirp generation. We define a base non-linear chirp as a monotonic curve growing from $(0, -\frac{BW}{2})$ to $(\frac{2^{SF}}{BW}, \frac{BW}{2})$, where the coordinate (x, y) represents the (time, frequency) boundary of this chirp. Since a monotone non-linear function can be approximated by the sum of a set of

polynomial functions in time-frequency domain, the base non-linear chirp thus can be represents as:

$$f_c(t) = \sum_{i=0}^{n} k_i t^i, t \in [0, \frac{2^{SF}}{BW}], f_c(t) \in [-\frac{BW}{2}, \frac{BW}{2}] \quad (2)$$

where $k_i, i \in [0, n]$ are a set of coefficients to fit the non-linear curve into the range of symbol time and BW. Notice that for a linear chirp, all these coefficients are zero except for $k_0 = -\frac{BW}{2}$ and $k_1 = \frac{BW^2}{2^{SF}}$.

To facilitate the coefficient configuration in different BW and SF settings, we further design a polynomial chirp function in a unified space $([0,1]_x \times [0,1]_{f(x)})$ as follow:

$$f(x) = \sum_{i=0}^{n} a_i x^i, x \in [0,1], f(x) \in [0,1] \quad (3)$$

where $a_i, i \in [0, n]$ is the ith coefficient. The relationship between the coefficient defined in the unified space and that defined in the time-frequency domain (i.e., k_i in Equation 2) can be represented as follows:

$$k_0 = BW \times a_0 - \frac{BW}{2}, \; k_i = \frac{BW^{i+1}}{2^{SF \times i}} a_i, i \in [1, n] \quad (4)$$

Given BW and SF, we can compute the coefficient k_i for each polynomial term defined in Equation 2 and generate the base up-chirp accordingly. The down-chirp can be generated by conjugating the base up-chirp.

Base non-linear chirp modulation. Once we have a base non-linear chirp, the transmitter then varies the initial frequency offset of this base chirp to modulate data:

$$h(t) = e^{j2\pi(f_0 + f_c(t))t} \quad (5)$$

where f_0 is the initial frequency offset of this chirp. In essence, given the same BW and SF configurations, CurvingLoRa achieves the same link throughput with the standard linear-chirp based LoRa.

Modulation knobs. By using different polynomial functions defined in the unified space (e.g., $f(x) = x$, $f(x) = x^2$ and $f(x) = 2x - x^2$), the transmitter can easily build different base chirps. Figure 10 shows a convex and a concave non-linear chirp produced by two different polynomial functions. These different polynomial functions provide us another knob to boost the throughput of concurrent LoRa transmissions. To understand the rationale behind this, let's consider a case where two LoRa transmissions (i.e., S_A and S_B) are happenly well-aligned at the receiver. Let S_R be their superposition.

• **Case one**: when both S_A and S_B are linear chirps, we are expected to see two separate energy peaks on FFT bins (shown in Figure 11(a)). In this case, all existing parallel decoding approaches [13,20,42,47,53,55,56,59] fail to disambiguate the collision symbols as these two well-aligned symbols exhibit similar FFT peaks.
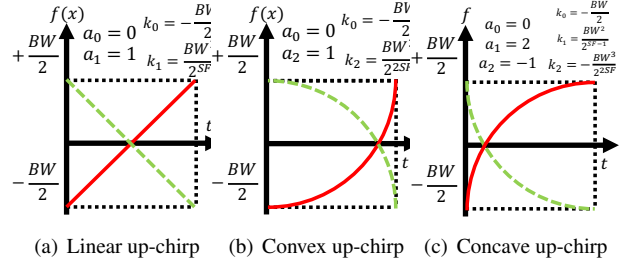
(a) Linear up-chirp (b) Convex up-chirp (c) Concave up-chirp

Figure 10: An illustration of linear and non-linear chirps with the corresponding function parameters.
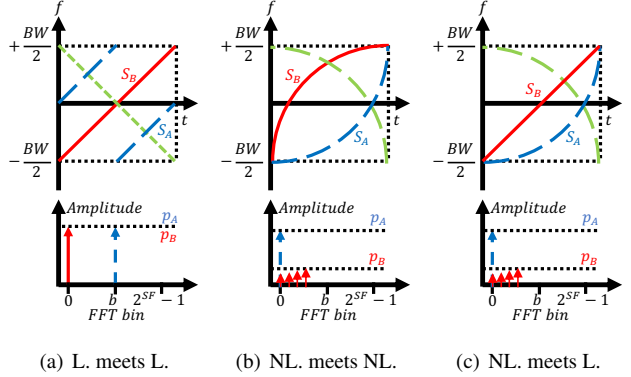
(a) L. meets L. (b) NL. meets NL. (c) NL. meets L.

Figure 11: An illustration of symbol collisions. (a): A linear chirp (L.) collides with another linear chirp. (b): A non-linear (NL.) chirp collides with another non-linear chirp. (c): A non-linear chirp collides with a linear chirp.

• **Case two**: when both S_A and S_B are non-linear chirps (i.e., generated by two different polynomial functions), the receiver can decode each symbol from their collision as follows. The receiver first multiplies S_R with the conjugate of S_A. As a result, the energy of S_B will be spread over multiple FFT bins, whereas the energy of S_A will concentrate on a single, isolated FFT bin, as shown in Figure 11(b). The receiver can easily pick up this energy peak and decode S_A. S_B can be decoded by replacing the down-chirp with the conjugate of S_B.

• **Case three**: when one of the collision symbols is based on linear chirp and another is based on non-linear chirp, the receiver can alternate between different down-chirps to decode each of them accordingly, as shown in Figure 11(c).

We have three takeaways from the above analysis: *i*) the transmitters can use different types of non-linear chirps as an orthogonal approach to boost the concurrency of LoRa transmissions. *ii*) the non-linear chirp based LoRa nodes can co-exist with those linear-chirp based legacy LoRa nodes. *iii*) the adoption of different non-linear chirps also facilitates the demodulation of well-aligned collision symbols.

5.2 Demodulation

Similar to linear chirp demodulation, the receiver operates dechirp to demodulate non-linear chirps.

Accounting for the Misalignment. Symbol alignment is crit-
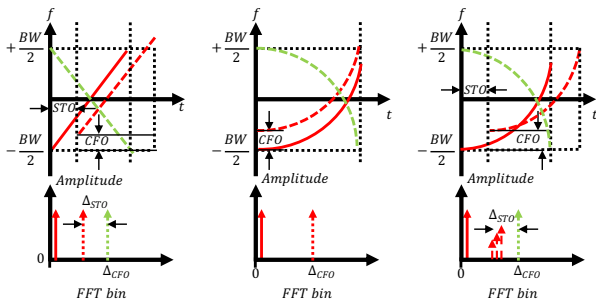
(a) with STO and CFO   (b) with CFO   (c) with STO and CFO

Figure 12: The influence of symbol time offset (STO) and carrier frequency offset (CFO) on demodulation. (a) both STO and CFO move the energy peak of a linear chirp from its desired FFT bin. (b) CFO moves the energy peak of a non-linear chirp from its desired FFT bin. (c) STO spreads the power of a non-linear chirp into multiple FFT bins.

ical to the demodulation performance, particularly for non-linear chirp demodulation, as the misalignment will spread the spectrum power of a chirp symbol into multiple frequency points, which fails the demodulation. While this misalignment, in theory, is only caused by the symbol time offset (STO) between the incident chirp symbol and the down-chirp, in practice, it is also affected by the carrier frequency offset (CFO) caused by clock offset.

In linear chirp demodulation, the dechirp converges the spectrum power of each linear chirp symbol to a specific frequency point. The existence of STO and CFO both renders the energy peak merely deviates from its desired position in FFT bins. After the dechirp, the receiver can thus leverage the preamble to estimate such frequency shift and then correct the symbol by applying the estimated frequency shift to the energy peak. However, such a post-processing approach cannot be directly applied to non-linear chirp, as the existence of STO will instead spread the spectrum energy into multiple FFT bins. Hence the receiver has to align the chirp symbol with the down-chirp and compensate for the CFO before operating dechirp on each non-linear chirp symbol.

To better understand this issue, we take Figure 12 as an example, where the receiver demodulates the linear chirp and non-linear chirp, respectively. To align the incoming linear chirp shown in Figure 12(a), the LoRa receiver operates multiplication on these two chirps. Due to the symbol time offset, the resulting FFT peak will be shifted from its desired bin by the amount of Δ_{STO}. CFO leads to an extra shift of the FFT peak Δ_{CFO}. By leveraging the preamble in the LoRa header, the receiver can easily estimate $\Delta_{CFO}+\Delta_{STO}$ and offset their impact on the energy peak. In contrast, the multiplication of two misaligned non-linear chirps (shown in Figure 12(b)) spreads the spectrum energy into multiple FFT bins, as shown in Figure 12(b). The existence of CFO further shift these FFT peaks and complicate the symbol alignment, as shown in Figure 12(c).
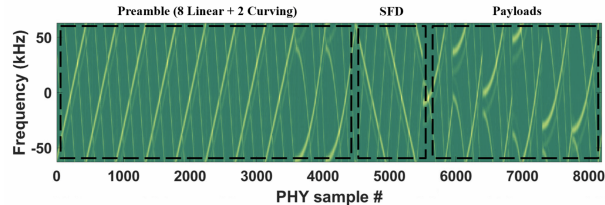

Figure 13: Packet format of CurvingLoRa.

In CurvingLoRa, we put a pair of conjugate chirps—a standard linear up-chirp followed by a standard linear down-chirp—as the pilot symbols of a LoRa packet to estimate the STO and CFO. Suppose these two linear chirps are well aligned with their conjugate counterpart in dechirping process, respectively. The resulting two FFT peaks are supposed to be superimposed at the same FFT bin without CFO. On the contrary, the existence of STO and CFO will shift these two FFT peaks by the amount of $\Delta_{CFO}+\Delta_{STO}$ and $-(\Delta_{CFO}+\Delta_{STO})$ from their desired position. The receiver then estimates the STO and CFO using the similar method as in NScale [47] and offsets the symbol misalignment and carrier frequency offset accordingly. It then operates dechirp on the corrected symbols to demodulate each symbol.

5.3 Frame Format

A typical LoRa packet comprises multiple preamble symbols, two mandatory sync word symbols, 2.25 Start Frame Delimiter (SFD) symbols followed by a variable number of payload symbols [28, 47]. Following the standard LoRa packet format, we encode the sync word symbols and payloads with non-linear chirps while retaining the linear chirps in preambles and SFDs, shown in Figure 13. The preamble contains eight identical linear up-chirps for packet detection and alignment, followed by two non-linear chirps of sync word for configuration recognition of payloads. The SFD consists of 2.25 standard down-chirps while the payload contains multiple chirp symbols with configurable length and chirp type. As mentioned in §5.2, a pair of up-chirp and down-chirp (i.e., pilot symbols) is needed to facilitate the symbol alignment. Instead of putting an extra pair of such pilot symbols on the LoRa packet, we reuse the last linear up-chirp symbol in the LoRa preamble and the first linear down-chirp symbol of SFD as the pilots. The use of linear chirp-based preamble may introduce the following two types of collisions:

● Linear chirps collide with non-linear chirps when the preamble of one packet happenly aligns with the payload of another packet. In this case, the receiver can still leverage the energy scattering and converging effect to detect the preamble and further demodulate each signal (§5.1).

● Linear chirps collide with linear chirps when the preamble of one packet happenly aligns with the preamble of another packet. In practice, however, this case rarely happens as the preamble contains only eight symbols, whereas the payload may last for hundreds of symbols [42, 47].

6 Implementation

Hardware and software. We implement CurvingLoRa on software-defined radios USRP N210 equipped with a UBX daughter board. The modulation and demodulation are implemented based on UHD+GNURadio [35]. The transmitter and receiver work on the 904.0MHz ISM band, equipped with a VERT900 antenna [3]. By default, the SF and BW are set to 10 and 125 kHz, respectively. The sampling rate is 1 MHz.

Experiment setups. We conduct trace-driven emulations to evaluate our system. Specifically, we fix the gateway's location and move a transmitter to different sites. The transmitter sends packets in different SFs and chirp symbol settings at each site. We then align LoRa traces collected from different sites with varying symbol offset offline to emulate packet collisions. The reasons are twofold. First, the trace-driven emulation allows us to manipulate symbol collisions in a fine-grained manner. It enables comprehensive collision settings, including various SNRs, SIR, and offsets to evaluate CurvingLoRa's performance. Second, it allows us to rapidly scale up the network size for concurrent transmission testing. Experiment setups are detailed in Appendix B.

Large-scale packet collision emulation. Due to the temporal diversity (e.g., the cars passing by may block the LoS path or generate a new reflection path), the LoRa traces collected from each site experience significantly different channel variations. This allows us to emulate large-scale LoRa networks by reusing each LoRa trace from a new LoRa transmitter. We further enhance the link diversity by varying the SIR of each trace at the gateway. The symbol offset is randomly chosen from $[0.2, 0.8] \times T_{symbol\_time}$.

Evaluation Metrics. We adopt three metrics to evaluate CurvingLoRa. i): *Symbol Error Rate (SER)* measures the demodulation of CurvingLoRa at the symbol level, under various SNRs and SIRs [42, 47]; ii): *Packet Delivery Rate (PDR)* computes the packet reception rate. in which 80% of symbols can be decoded successfully.[5] iii): *Throughput* can be derived with the received packets and decoded symbols, denoted by Symbol/Second. Note that LoRa gateways are usually deployed with tethered power supplies, and thus we do not consider energy consumption at the gateway [42, 47].

Baselines. We compare our design with two SOTA LoRa collision decoding systems mLoRa [53] and NScale [47]. These two systems represent two mainstreaming designs, namely, successive interference cancellation [22, 41, 53] and spectral energy based approach [13, 20, 42, 47]. The standard LoRaWAN is also adopted for comparison. As a proof of concept, we design four types of non-linear chirps to evaluate:

(1): *quadratic*1—$f(t) = t^2$	(2): *quadratic*2—$f(t) = -t^2 + 2t$
(3): *quartic*1—$f(t) = t^4$	(4): *quartic*2—$f(t) = -t^4 + 4t^3 - 6t^2 + 4t$

---

[5]Most error correction codes can recover 1/5 symbol errors [48].

7 Evaluation

We present the results in this section. §7.1 first compares CurvingLoRa with linear chirps at the symbol and packet level, followed by the outdoor experiments at the campus scale in §7.2. Finally, we provide the large-scale emulation to explore the impact of concurrency on CurvingLoRa in §7.3. And indoor evaluations can be found in §C.

7.1 Overall Comparisons with Linear Chirps

Noise resilience. We compare the noise resilience of CurvingLoRa with LoRaWAN in the presence of collisions. Figure 14(a)-(c) shows the SER in various SNR and SF settings. When *SF*=8, we observe that both LoRaWAN and four types of non-linear chirps fail to demodulate packets in extremely low SNR conditions (i.e., SNR≤-25dB). As the SNR grows to −15dB, the SER achieved by non-linear chirps drops dramatically to around 1%, whereas the SER of linear chirps is still above 20%. As the SNR grows further, we observe the SER of non-linear chirps is always 10× lower than that of the linear chirps, e.g., 0.3% versus 3% at SNR=30dB. Similar trends hold for *SF*=10 and 12.

We also evaluate the impact of narrow-band interference (i.e., RFID) on CurvingLoRa symbol decoding (Appendix D). We observe these different types of non-linear chirps can achieve descent resilience to narrow-band interference. We also find that the noise resilience of non-linear chirps varies with the chirp shape, and we leave the non-linear chirp selection as our future work. In addition, the evaluation results verify the Gaussian noise resilience observed by our analysis (§4.3) in Appendix E.

Symbol offset. Next, we compare the SER of CurvingLoRa and linear chirps in various symbol offset settings. Specifically, from our collected dataset, we randomly pick up LoRa symbols with different SFs (SF=8,10,12) and SNRs ([−15dB,15dB]). We then vary the symbol offset between two collision symbols from 10% to 50% and plot their SER in Figure 14(d). In consistency with our simulation in Figure 6, we observe the SER of linear chirps drops with increasing symbol offsets. In contrast, the SER achieved by non-linear chirps maintains a low level, with the maximal value of 1% when the offset of two collision symbols is merely 10%. In contrast, the linear chirp's SER varies from 1% to 80% as the symbol offset decreases. These results demonstrate that the non-linear chirps are robust to collisions with different symbol offsets.

Resolving near-far issue. We also compare CurvingLoRa with linear chirp-based LoRa (i.e., LoRaWAN) in the presence of the near-far issue. In particular, we vary the SIR of the targeting symbol and measure its SER in each SIR setting. Figure 15(a)-(c) show the results in three different SF settings. We observe the standard LoRaWAN fails to decode weak targeting signal (i.e., SIR<0dB) across all three *SF* settings.
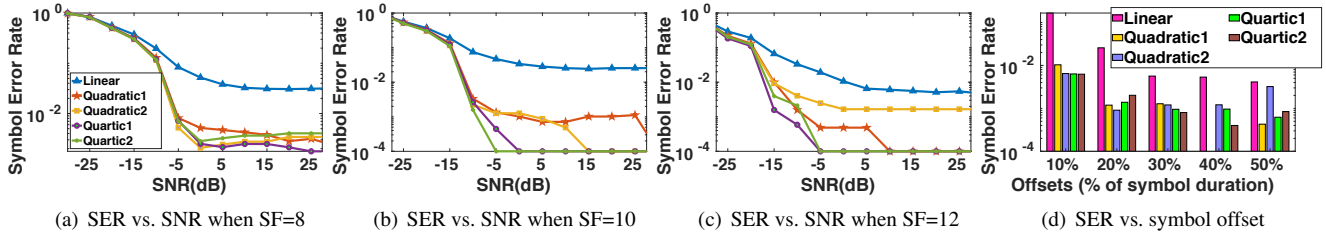

(a) SER vs. SNR when SF=8 (b) SER vs. SNR when SF=10 (c) SER vs. SNR when SF=12 (d) SER vs. symbol offset

Figure 14: Linear vs non-linear: symbol error rate ($SIR \approx 0dB$) in different SFs, SNRs and symbol offsets settings.
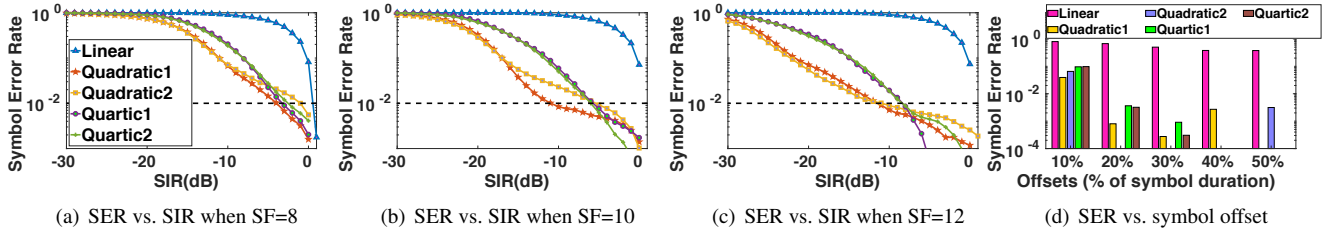

(a) SER vs. SIR when SF=8 (b) SER vs. SIR when SF=10 (c) SER vs. SIR when SF=12 (d) SER vs. symbol offset

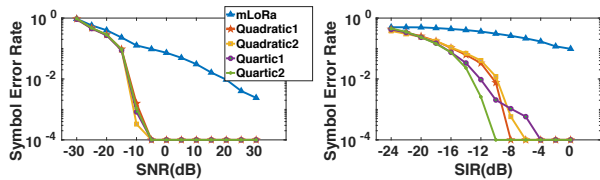Figure 15: Linear vs. non-linear: symbol error rate ($SNR > 30dB$) in different SF, SIR, and symbol offset settings.

Figure 16: Head-to-head comparison with mLoRa [53].

In contrast, by leveraging the power scattering effect, all four types of non-linear chirps in CurvingLoRa can successively demodulate weak symbols in the presence of strong collisions. For instance, the averaging SIR threshold for achieving less than 1% SER is -3.7dB, $-7.7dB$, and $-10.2dB$ for SF=8, 10, and 12, respectively. We further vary the symbol offset of two collision symbols and measure the SER achieved by CurvingLoRa and LoRaWAN. The SIR and SF of symbols for evaluation varies from -10dB to 1dB, and from 8 to 12, respectively. The results are shown in Figure 15(d). We observe CurvingLoRa achieves a robust low SER (i.e., less than 1%) in the presence of a large symbol offset. It then degrades slightly as the symbol offset decreases. In contrast, the linear chirp achieves consistently high SER (i.e., $\geq$38%) in all five different symbol offset settings. These results clearly demonstrate that CurvingLoRa can successfully decode weak signals in the presence of strong collisions in various conditions.

Head-to-head comparison with mLoRa [53]. We compare CurvingLoRa with mLoRa on using our indoor dataset. Figure 16(a) shows the SER of each system in the presence of two collision packets. We observe that the SER of mLoRa drops gradually from 15% to 1% as the SNR grows from -15dB to 15dB. It finally drops to 0.3% when SNR grows to 30dB. In contrast, all four types of non-linear chirps adopted by CurvingLoRa achieve a consistently low SER ($\leq$0.01%) when SNR is larger than -15dB. Similarly, as shown in Figure 16(b), the SER of mLoRa is over 10% in the presence of

a strong collision (i.e., $SIR < 0dB$) whereas CurvingLoRa can demodulate weak signals at an $SER < 1\%$ as long as the SIR is larger than -12dB.

Remarks. These experiments show the advantage of CurvingLoRa in dealing with near-far issues. It also reveals that the performance of CurvingLoRa varies with the non-linear function being adopted. Overall the quadratic function $f(t) = t^2$ achieves consistently better SER than the other types of non-linear functions. We leave the exploration of non-linear space as our future work.

7.2 Concurrency at the Campus Scale

We evaluate CurvingLoRa on decoding collisions in different numbers of concurrent transmissions (termed as N) settings. In particular, we measure the SER, PDR, and network throughput and compare with mLoRa [53], NScale [47], and LoRaWAN three baselines. Finally, we repeat the experiments in indoor environments and put the results in Appendix C.

As N grows, the SERs of LoRaWAN, mLoRa, and NScale all increase gradually, as shown in Figure 17(a). Specifically, LoRaWAN can only demodulate the strongest transmission for most settings. And mLoRa achieves a slightly better performance than LoRaWAN. However, its SER aggravates significantly ($\geq$50%) when demodulating more than four concurrent transmissions. Besides, NScale performs better than the above two schemes, and the SER grows gradually from less than 15% to 55% when N grows to 12. In contrast, CurvingLoRa achieves an average SER of less than 25% in all settings.

Figure 17(b) shows the packet delivery ratio achieved by these systems. We observe that as N grows, the PDR achieved by CurvingLoRa drops slightly from 100% to 65%. In contrast, the PDR drops significantly to less than 36.5%, 25.0%, and 12.5% for NScale, mLoRa, and LoRaWAN, respectively. We further compute the network throughput and plot the results in Figure 17(c). The overall network throughput of CurvingLoRa,
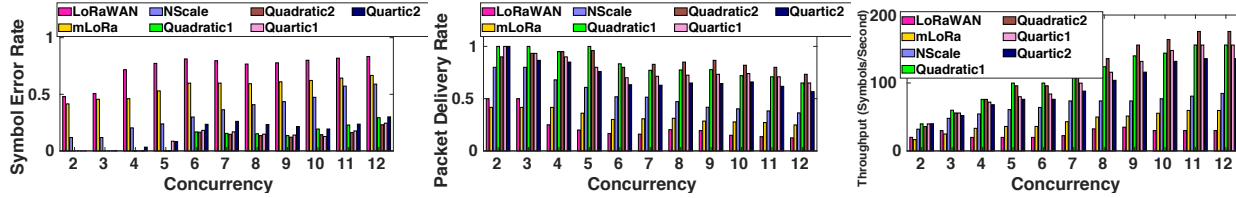
Figure 17: Outdoor experiment: examine the impact of concurrent transmissions on SER, PDR, and network throughput.
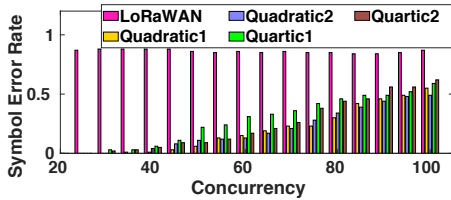

Figure 18: Emulation of large-scale collisions.

NScale, and mLoRa grow with the increasing N. However, the network throughput of LoRaWAN manifests a converse trend due to the magnified interference as N grows. Taking further scrutiny on this result, we find that the network throughput of CurvingLoRa grows almost linearly with N. In contrast, the growing trend of network throughput in both NScale and mLoRa drops gradually as N grows. This is because the near-far issue grows extensively with increasing concurrent transmissions. However, both NScale and mLoRa are not scaling to such circumstances. Statistically, when N=12, the average network throughput of CurvingLoRa is $5.21\times$, $2.61\times$, and $1.84\times$ higher than that of LoRaWAN, mLoRa, and NScale.

We also evaluate CurvingLoRa's performance in the wild using three USRPs (two as transmitters, and another one as the receiver). We vary the transmission power to manipulate the signal SIR (from 0 to -8dB) and calculate the SER in different SIR settings. The results show that CurvingLoRa can achieve consistently low SER across four different types of non-linear chirps (Appendix F).

7.3 Large-scale emulation

We also emulate large-scale collisions using the data collected both indoors and outdoors. Specifically, in each number of concurrent transmission settings, we only measure the SER of the weakest transmissions as those stronger transmissions are likely to be correctly demodulated. Figure 18 shows the SER of CurvingLoRa and LoRaWAN when the SIR varies randomly between $[-5dB, 0dB]$, We observe that all four types of non-linear chirps adopted by CurvingLoRa can successively demodulate the weakest transmission (i.e., SER=0) when the number of concurrent transmissions is less than 30. The SER then grows up gradually as the network scales. It peaks at 50% when 100 transmitters work concurrently. In contrast, the standard LoRaWAN fails to demodulate the weakest transmission with more than two concurrent transmissions.

8 Limitation and Future Work

We discuss the limitations of current design and evaluation that may shed light on future research.

Non-linear Chirp Selection. CurvingLoRa's performance gain on collision symbol decoding is determined by the spectral energy distribution of the interfered chirp symbols. Our experiment results show that such performance gain varies among different types of non-linear chirps, leaving rooms for further exploration.

Deployment and evaluation. Our experiments are largely based on emulation, and thus may not reflect the impact of channel dynamics on packet demodulation. Future works may focus on building a CurvingLoRa test-bed for long-term system evaluation.

Backward compatibility. CurvingLoRa node can generate standard linear chirps (§5.1). Its demodulation can be easily adapted to commodity LoRa gateways by replacing the standard linear down-chirp with its non-linear counterpart (§5.3). In addition, like standard LoRa networks, CurvingLoRa can adopt unslotted ALOHA protocol as its MAC-layer. Therefore, we expect CurvingLoRa can co-exist with existing LoRa network. An interesting direction worth exploring is to examine whether the recent innovations on LoRa PHY-layer and MAC-layer [14, 15, 18] are applicable to CurvingLoRa.

9 Conclusion

We have presented the design, implementation, and evaluation of CurvingLoRa, a PHY-layer amendment to LoRaWAN. By replacing the linear-chirp modulation on standard LoRaWAN with its non-linear chirp counterpart, the receiver can effectively demodulate large numbers of collided LoRa transmissions in extreme SNR, SIR, and symbol offset conditions. We practice this idea by designing a holistic PHY layer and implementing it on software-defined radios. The results demonstrate CurvingLoRa improves the network throughput by $7.6\times$ against the standard LoRaWAN, outperforming two state-of-the-art approaches by $1.6\times$ and $2.8\times$, respectively.

Acknowledgments

References

[1] N. Abramson. The aloha system: Another alternative for computer communications. *Proceedings of the November 17-19, 1970, fall joint computer conference*, 281–285, 1970.

[2] L. Alliance. Lorawan specification. https://lora-alliance.org/about-lorawan. Accessed 08-Apr-2020.

[3] V. Antenna. Vert900 vertical antenna (824-960 mhz, 1710-1990 mhz) dualband. *https://www.ettus.com/all-products/vert900/*, Retrieved by May 10th 2021.

[4] S. R. Benson. Modern digital chirp receiver: Theory, design and system integration, 2015.

[5] A. Berni, W. Gregg. On the utility of chirp modulation for digital signaling. *IEEE Transactions on Communications*, 1973.

[6] F. Calì, M. Conti, E. Gregori. Dynamic ieee 802.11: design, modeling and performance evaluation. *Proceedings of the International Conference on Research in Networking*. Springer, 2000.

[7] M. Centenaro, L. Vangelista, A. Zanella, M. Zorzi. Long-range communications in unlicensed bands: The rising stars in the iot and smart city scenarios. *IEEE Wireless Communications*, 2016.

[8] J. Chan, A. Wang, A. Krishnamurthy, S. Gollakota. DeepSense: Enabling Carrier Sense in Low-Power Wide Area Networks Using Deep Learning. *arXiv:1904.10607 [cs]*, 2019.

[9] Y. Cheng, H. Saputra, L. M. Goh, Y. Wu. Secure smart metering based on lora technology. *Proceedings of IEEE International Conference on Identity, Security, and Behavior Analysis (ISBA)*, 2018.

[10] C. Chiu, Z. Zhang, L. T. Hsien. The near/far effect in local aloha radio communications. *IEEE Journal of Solid-State Circuit*, 2020.

[11] U. Coutaud, M. Heusse, B. Tourancheau. Fragmentation and Forward Error Correction for LoRaWAN small MTU networks. *Proceedings of EWSN*. Junction Publishing, 2020.

[12] A. W. Doerry. Generating nonlinear fm chirp waveforms for radar. Tech. rep., Sandia National Laboratories, 2006.

[13] R. Eletreby, D. Zhang, S. Kumar, O. Yağan. Empowering low-power wide area networks in urban settings. *Proceedings of ACM SigComm*, 2017.

[14] A. Gadre, R. Narayanan, A. Luong, A. Rowe, B. Iannucci, S. Kumar. Frequency configuration for low-power wide-area networks in a heartbeat. *Proceedings of USENIX NSDI*, 2020.

[15] A. Gamage, J. C. Liando, C. Gu, R. Tan, M. Li. LMAC: efficient carrier-sense multiple access for LoRa. *Proceedings of ACM MobiCom*, 2020.

[16] S. Gollakota, S. D. Perli, D. Katabi. Interference alignment and cancellation. *Proceedings of the ACM SIGCOMM*, 2009.

[17] J. Haxhibeqiri, A. Karaagac, F. Van den Abeele, W. Joseph, I. Moerman, J. Hoebeke. Lora indoor coverage and performance in an industrial environment: Case study. *Proceedings of IEEE international conference on emerging technologies and factory automation (ETFA)*, 2017.

[18] M. Hessar, A. Najafi, S. Gollakota. Netscatter: Enabling large-scale backscatter networks. *Proceedings of USENIX NSDI*, 2019.

[19] N. Hosseini, D. W. Matolak. Nonlinear quasi-synchronous multi user chirp spread spectrum signaling. *IEEE Transactions on Communications*, 2021.

[20] B. Hu, Z. Yin, S. Wang, Z. Xu, T. He. Sclora: Leveraging multi-dimensionality in decoding collided lora transmissions. *Proceedings of IEEE ICNP*, 2020.

[21] T. Instruments. Optimizing power consumption and power-up overshoot using TPS54160-Q1 family in automotive applications. https://www.ti.com/lit/an/slva436a/slva436a.pdf?ts=1622474296492&ref_url=https%253A%252F%252Fwww.google.com.hk%252F. Accessed 30-May-2021.

[22] R. Jung, P. Levis. Receiving colliding lora packets with hard information iterative decoding. *Proceedings of IEEE GLOBECOM*, 2021.

[23] M. A. Khan, R. K. Rao, X. Wang. Performance of quadratic and exponential multiuser chirp spread spectrum communication systems. *Proceedings of IEEE SPECTS*, 2013.

[24] K.-R. Kim, S. Kim, C.-H. Ki, T.-H. Kim, H. Yang, J.-H. Kim. Development and comparison of DDS and multi-DDS chirp waveform generator. *Proceeding of IEEE International Geoscience and Remote Sensing Symposium (IGARSS)*, 2019.

[25] C. Lesnik, A. Kawalec. Modification of a weighting function for nlfm radar signal designing. *Acta Physica Polonica A*, 2008.

[26] C. Li, Z. Cao. Lora networking techniques for large-scale and long-term iot: A down-to-top survey. *ACM Computing Surveys*, 2022.

[27] C. Li, H. Guo, S. Tong, X. Zeng, Z. Cao, M. Zhang, Q. Yan, L. Xiao, J. Wang, Y. Liu. Nelora: Towards ultra-low snr lora communication with neural-enhanced demodulation. *Proceedings of ACM Sensys*, 2021.

[28] J. C. Liando, A. Gamage, A. W. Tengourtius, M. Li. Known and unknown facts of lora: experiences from a large-scale measurement study. *ACM Transactions on Sensor Networks*, 2019.

[29] L. Lou, Z. Fang, B. Chen, T. Guo, Z. Liu, Y. Zheng. A DDS-driven ADPLL chirp synthesizer with ramp-interpolating linearization for FMCW radar application in 65nm CMOS. *Proceeding of IEEE International Symposium on Circuits and Systems (ISCAS)*, 2018.

[30] D. Magrin, M. Centenaro, L. Vangelista. Performance evaluation of lora networks in a smart city scenario. *Proceeding of IEEE International Conference on Communications (ICC)*, 2017.

[31] P. Marcelis, N. Kouvelas, V. S. Rao, V. Prasad. DaRe: Data Recovery through Application Layer Coding for LoRaWAN. *IEEE Transactions on Mobile Computing*, 2020.

[32] B. Mohring, C. Moroder, U. Siart, T. Eibert. Broadband, fast, and linear chirp generation based on DDS for FMCW radar applications. *Proceeding of IEEE Radar Conference (RadarConf)*, 2019.

[33] M. Mollanoori, M. Ghaderi. Uplink scheduling in wireless networks with successive interference cancellation. *IEEE Transactions on Mobile Computing*, 2013.

[34] J. Navarro-Ortiz, S. Sendra, P. Ameigeiras, J. M. Lopez-Soler. Integration of lorawan and 4g/5g for the industrial internet of things. *IEEE Communications Magazine*, 2018.

[35] G. R. project. Gnu radio website. http://www.gnuradio.org. Accessed 07-Apr-2020.

[36] F. provider. Zynq-7000 fpga. https://www.renesas.com/sg/en/application/technologies/fpga-power/zynq-7000. Accessed 30-May-2021.

[37] Q. M. Qadir, T. A. Rashid, N. K. Al-Salihi, B. Ismael, A. A. Kist, Z. Zhang. Low Power Wide Area Networks: A Survey of Enabling Technologies, Applications and Interoperability Needs. *IEEE Access*, 2018.

[38] A. Research. Nb-iot and lte-m issues to boost lora and sigfox near and long-term lead in lpwa network connections. *https://tinyurl.com/2026- cellular-iot*, Retrieved by Nov 19th 2020.

[39] A. Rokita. Direct analog synthesis modules for an X-band frequency source. *Proceeding of 12th International Conference on Microwaves and Radar*, 1998.

[40] Y. Sangar, B. Krishnaswamy. WiChronos: energy-efficient modulation for long-range, large-scale wireless networks. *Proceedings of ACM MobiCom*, 2020.

[41] M. O. Shahid, M. Philipose, K. Chintalapudi, S. Banerjee, B. Krishnaswamy. Concurrent interference cancellation: decoding multi-packet collisions in lora. *Proceedings of ACM SIGCOMM*, 2021.

[42] T. Shuai, X. Zhenqiang, W. Jiliang. Colora: Enable muti-packet reception in lora. *Proceedings of IEEE INFOCOM*, 2020.

[43] R. S. Sinha, Y. Wei, S.-H. Hwang. A survey on lpwa technology: Lora and nb-iot. *Ict Express*, 2017.

[44] J. R. Smith, A. P. Sample, P. S. Powledge, S. Roy, A. V. Mamishev. A wirelessly-powered platform for sensing and computation. *Proceedings of ACM UbiComp*, 2006.

[45] K. Staniec, M. Kowal. Lora performance under variable interference and heavy-multipath conditions. *Wireless communications and mobile computing*, **2018**, 2018.

[46] V. Talla, M. Hessar, B. Kellogg, A. Najafi, J. R. Smith, S. Gollakota. Lora backscatter: Enabling the vision of ubiquitous connectivity. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 2017.

[47] S. Tong, J. Wang, Y. Liu. Combating packet collisions using non-stationary signal scaling in LPWANs. *Proceedings of ACM MobiSys*, 2020.

[48] D. Tse, P. Viswanath. *Fundamentals of wireless communication*. Cambridge university press, 2005.

[49] D. tutorials. Fundamentals of direct digital synthesis (DDS). https://www.analog.com/media/en/training-seminars/tutorials/MT-085.pdf. Accessed 30-May-2021.

[50] N. Varsier, J. Schwoerer. Capacity limits of lorawan technology for smart metering applications. *Proceedings of IEEE International Conference on Communications (ICC)*, 2017.

[51] R. Venkatesha. p-CARMA: Politely Scaling LoRaWAN†. *Proceedings of EWSN*. Junction Publishing, 2020.

[52] Q. Wang. *Non-Linear Chirp Spread Spectrum Communication Systems of Binary Orthogonal Keying Mode*. Ph.D. thesis, The University of Western Ontario, 2015.

[53] X. Wang, L. Kong, L. He, G. Chen. mlora: A multi-packet reception protocol in lora networks. *Proceedings of IEEE ICNP*, 2019.

[54] R. Want. An introduction to rfid technology. *IEEE pervasive computing*, **5**(1), 25–33, 2006.

[55] X. Xia, Y. Zheng, T. Gu. Ftrack: parallel decoding for lora transmissions. *Proceedings of ACM Sensys*, 2019.

[56] Z. Xu, S. Tong, P. Xie, J. Wang. FlipLoRa: Resolving Collisions with Up-Down -Orthogonality. *Proceedings of IEEE International Conference on Sensing, Communication, and Networking (SECON)*, 2020.

[57] H. Yang, S.-B. Ryu, H.-C. Lee, S.-G. Lee, S.-S. Yong, J.-H. Kim. Implementation of DDS chirp signal generator on FPGA. *Proceeding of International Conference on Information and Communication Technology Convergence (ICTC)*, 2014.

[58] Y. Yao, Z. Ma, Z. Cao. Losee: Long-range shared bike communication system based on lorawan protocol. *Proceedings of ACM EWSN*, 2019.

[59] W. Zhe, K. Linghe, X. Kangjie, H. Liang, W. Kaishun, C. Guihai. Online concurrent transmissions at lora gateway. *Proceedings of IEEE INFOCOM*, 2020.

[60] D. Zorbas. Design Considerations for Time-Slotted LoRa(WAN). *Proceedings of EWSN*. Junction Publishing, 2020.

A   Energy Scattering Effect

We use the quadratic chirp (i.e., $f(t) = k_2 t^2 + k_0$ in Equation 2) as an example to explain the energy scattering effect of non-linear chirps. For the dechirp of a chirp symbol, the receiver multiplies it with the base corresponding down-chirp as follows:

$$e^{j2\pi(f_0+f_c(t+t_{gap}))t} * e^{-j2\pi f_c(t)t} = e^{j2\pi F(t)t} \tag{6}$$

where t_{gap} denotes the symbol offset between the incident chirp symbol and the FFT window (i.e., the base down-chirp); f_0 represents the initial frequency offset of this non-linear chirp. The spectral energy peak is determined by the term $F(t)$ for different types of chirps. For a linear chirp (i.e., $f(t) = k_1 t + k_0$), it can always focus on a single frequency point in the dechirp since $F(t) = f_0 + k_1(t+t_{gap}) + k_0 - (k_1 t + k_0) = f_0 + k_1 t_{gap}$ given a fixed t_{gap}. In contrast, it spreads the energy over a frequency bins as follows for a quadratic chirp:

$$F(t) = f_0 + k_2(t+t_{gap})^2 + k_0 - (k_2 t^2 + k_0)$$
$$= f_0 + k_2 t_{gap}^2 + 2k_2 t_{gap} \times t \tag{7}$$

When the incident chirp is not well aligned with the down-chirp (i.e., $t_{gap}! = 0$), from the above equation, we can find that the spectrum energy of this incident chirp will spread to multiple FFT bins. In contrast, when $t_g ap = 0$, we have $F(t) = f_0$, indicating the spectrum energy will converge a single frequency point f_0.

B   Experiment Setups

We evaluate CurvingLoRa with LoRa traces collected from two different environments:

Figure 19: The indoor experimental plan and SDR devices spread out across tens of rooms.
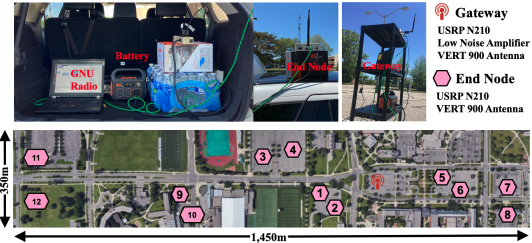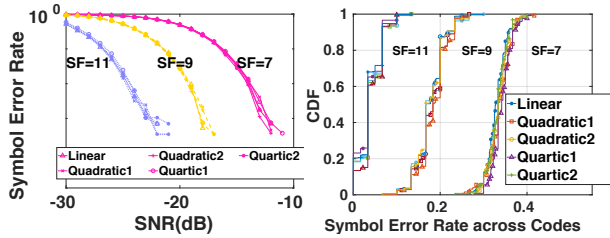

Figure 20: Bird view of the outdoor experiment field with the mobile gateway and LoRa nodes.

• **Indoor scenario**. We place transmitters and gateway on a 30.48m×21.34m office building. The offices are separated by concrete walls. Figure 19 shows the floor-plan of this office building. We place the gateway in the kitchen and move the transmitter to 10 different locations. Due to the blockage of walls, most LoRa transmissions are under the non-line-of-sight (NLoS) condition.

• **Outdoor scenario**. We deploy a campus-scale testbed outdoors. The gateway powered by a UPS is placed on the parking lot. We move a transmitter to 12 locations and collect LoRa transmissions in both LoS and NLoS conditions with various link distances. The bird view of the outdoor testbed is shown in Figure 20.

C   Concurrency in the Indoor Environment

Similar to the SER trend of outdoor experiments, we observe a huge SER gap between CurvingLoRa and its competitors as N grows in indoor experiments (Figure 22(a)). On the other hand, compared with outdoor experiments, we find that all indoor-space systems achieve slightly lower SER, with up to 7.75%-11.26% when N=10. This is because the transmitters are facing less severe near-far issues indoors. The packet delivery rate in indoor experiments shows a similar trend with their outdoor counterparts, as shown in Figure 22(b). Specifically, the PDR achieved by CurvingLoRa drops slightly from 100% to 87.10% on average as N grows from 2 to 10. While both mLoRa and LoRaWAN drop significantly from around 50.5% to less than 40.0% and 10.6%, respectively. Figure 22(c) shows the network throughput achieved by these three systems in an indoor environment. The overall network

(a) Overall SER performance   (b) SER distribution in code space

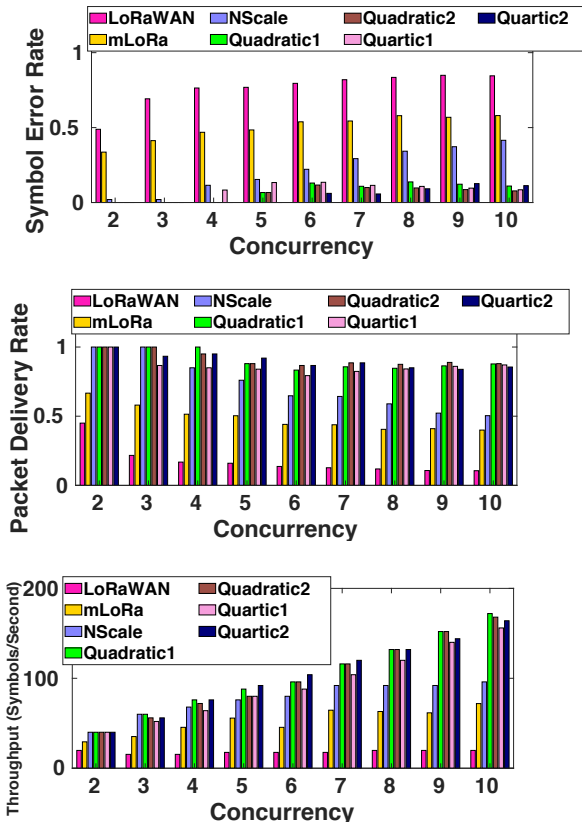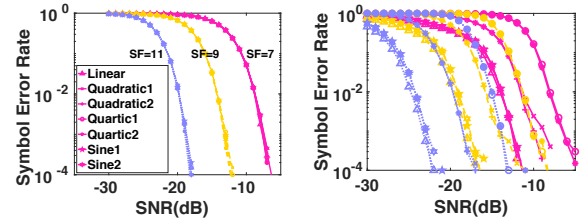Figure 21: Noise resilience in the absence of collisions.

(a) Under Gaussian noise.   (b) Under RFID narrow-band interference.

Figure 23: SER of various types of non-linear chirps under various noise sources.

(a) SER across positions.   (b) SER across SIRs.

Figure 24: Filed study for various types of non-linear chirps.

a WISP 5.0 [44] tag to generate RFID signals with the data rate of 10KHz, and manually superpose the RFID jamming with LoRa's non-linear transmissions. Finally, we evaluate the SER with controlled SNR levels.

Illustrated in Figure 23, we show the SER fluctuation over SNR levels under the Gaussian noise and narrow-band interference from RFID signals. First, compared to the Gaussian noise, the CSS mechanism with Sine and linear chirps achieves a higher resilience for the RFID narrow-band interference. Figure 23(a) shows that CSS with SF=11 (e.g., yellow lines) requires the SNR higher than $-20dB$ for all types of chirps to achieve the SER lower than %1. In contrast, the SNR threshold under the same configuration is $-25dB$ for the linear and Sine chirps under the RFID narrow-band interference as shown in Figure 23(b). Theoretically, the dechirp can alleviate the impact of interference by spreading its signal energy over the whole spectrum, enabling LoRa's long-range communication, especially against the narrow-band interference [45]. Second, the Sine chirps achieve the same interference resilience with the linear chirp across different SFs. In contrast, the Quadratic and Quartic chirps deliver a higher SER under the same configurations as shown in Figure 23(b). The reason is that the RFID signals are On-Off-Keying modulation, which is more similar with Quadratic and Quartic chirps than linear and Sine chirps. The dechirp processing picks the corresponding base down-chirp for the adopted chirp signals. As a result, Quadratic and Quartic base down-chirps cannot spread the RFID interference signals as well as linear and Sine base down-chirps do, resulting in less resilience for the RFID narrow-band interference. By studying the impact of different chirp types under the specific noise distribution in the wild, we can select the chirp types adaptively for reliable transmissions [19].

Figure 22: Indoor experiment: examine the impact of concurrent transmissions on SER, PDR, and network throughput.

throughput of CurvingLoRa grows consistently as the number of transmitters scales up. When ten packets collide simultaneously with significant power difference, the average network throughput of CurvingLoRa is about $1.6 \sim 7.6 \times$ higher than the network throughput achieved by NScale, mLoRa, and the standard LoRaWAN.

D   Impact of Narrow-band Interference

In this section, we use the RFID signal, a representative narrow-band signal, to study the impact of narrow-band interference on CurvingLoRa's performance. An RFID transceiver can communicate within the band at 902 to 928 MHz [54], overlapping with LoRa transmissions. Specifically, we control

E Impact of Gaussian Noise

We compare the noise resilience of CurvingLoRa with linear chirps under common Gaussian noise. The results are shown in Figure 21(a). Per our analysis (§4.3), we observe all four types of non-linear chirps in CurvingLoRa demonstrate comparable noise resilience with linear chirps across all three *SF* settings. Figure 21(b) shows that the symbol error rate is distributed evenly over the entire code space, confirming that the non-linear chirp achieves consistent SER for different symbols.

F Field Study for Collision Resolving

Setup. To evaluate CurvingLoRa in the wild, we deploy three USRPs at the campus-scale outdoor for the field study. Specifically, for each time, two are deployed to transmit LoRa packets at different locations simultaneously to produce the colli-

sions with one USRP as the receiver. We manually adjust the power of LoRa packets from these two USRP transmitters to control the SIR between them. For example, we deploy the two transmitters in six positions, with the position ID (e.g., 1, 3, 5, 7, 9) denoted in Figure 20. It covers four types of non-linear chirps for quadratic and quartic forms. Furthermore, the SIR is controlled to vary from 0 to -8*dB*, shown in Figure 24. As a result, we can evaluate CurvingLoRa's performance via the average SER of multiple packets across different locations and SIRs in the wild.

Results. Illustrated in Figure 24, CurvingLoRa's four types of non-linear modulation schemes perform consistently with our emulation for the field study. For example, its SER for two concurrent transmissions keeps lower than 1% for all locations except the furthest location #11 where the SIR is lower than other locations. Meanwhile, the SER increases as the SIR decreases from 0 to -8*dB*, with a larger power difference for these concurrent transmissions.

PLatter: On the Feasibility of Building-scale Power Line Backscatter

Junbo Zhang[1], Elahe Soltanaghai[2], Artur Balanuta[1], Reese Grimsley[1],
Swarun Kumar[1], and Anthony Rowe[1]

[1]*Carnegie Mellon University,* [2]*University of Illinois at Urbana-Champaign*

Abstract

This paper explores the feasibility of reusing power lines in a large industrial space to enable long-range backscatter communication between a single reader and ultra-low-power backscatter sensors on the walls that are physically not connected to these power lines, but merely in their vicinity. Such a system could significantly improve the data rate and range of backscatter communication with only a single reader installed, by using pre-existing power lines as communication media. We present PLatter, a building-scale backscatter system that allows ultra-low-power backscatter sensors or tags attached to walls with power lines right behind them to communicate with a reader several hundred feet away. PLatter achieves this by inducing and modulating parasitic impedance on power lines with the tag toggling between two loads in specialized patterns. We present a detailed evaluation of both the strengths and weaknesses of PLatter on a large industrial testbed with power lines up to 300 feet long, demonstrating a maximum data rate of 4 Mbps.

1 Introduction

This paper asks "Can we read ultra-low-power sensors in a large industrial or commercial building with a single reader using the power line system?" Given the significant cost associated with retrofitting an industrial building, a wired network for IoT installation is not desirable. On the other hand, long-range wireless networks are either power-hungry (e.g., WiFi or cellular), or support a very low data rate (e.g., LoRa). In this paper, we explore an alternative approach by combining backscatter and power line communication technologies. Backscatter systems [24, 16, 43, 12] are popular for their ultra-low power consumption, suitable for battery-free objects and low-power sensors. However, they are notorious for their short operation range (e.g., a few cm to 10 m). There has been some research on extending the range of backscatter
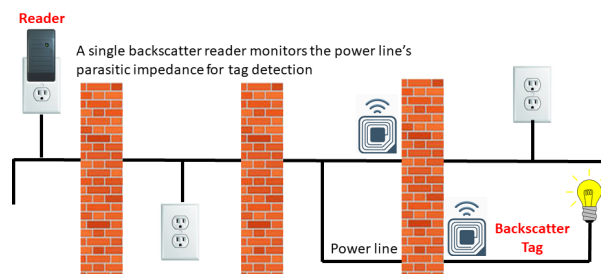

Figure 1: PLatter leverages the pre-existing power line infrastructure to provide long-range backscatter communication between backscatter tags and a single reader.

systems [42, 39, 23], but they either work only outdoors and in line-of-sight scenarios, or support a very low data rate. To address these limitations, this paper proposes to use the power line system, an existing framework that pervades nearly all buildings and spreads along the walls to every room, as a *wired-wireless medium* to read backscatter tags attached to the walls, thus enabling long-range, high data rate backscatter with a single reader.

Consider an industrial IoT context where backscatter sensors, powered by coin-cell batteries, monitor lighting, temperature, or fault conditions, and can be conveniently attached to the closest available walls or ceilings, just a few cm away from the ubiquitous power lines passing behind. A single reader plugged into an outlet can then read their data even when they are way out of the wireless communication range or significantly obstructed. There exists a rich literature in using power lines for communication [10, 45, 6, 41], positioning [31, 48], synchronization [34, 44, 35], sensing [20, 29, 30, 9, 14, 8], or as a source to harvest stray electromagnetic energy [21, 19]. One naive approach is to directly attach the IoT sensors to power outlets and use traditional power line communication. However, this limits the number of sensors and their locations to only a few outlets available in each room. Instead, we leverage existing power lines for backscatter communication by attaching a single reader to the power line and placing ultra-low-power tags any-

where along the power cables.

This paper explores the feasibility of such a building-scale backscatter system (Fig. 1), PLatter[1], where the power lines enable a single reader to receive sensory data from multiple ultra-low-power tags that are well beyond its RF communication range. PLatter's design leverages fluctuations in the parasitic impedance induced to the power line system by the backscatter tags as they toggle between two loads. In principle, conductors close to the power lines give rise to parasitic impedance; it is usually unwanted, though unavoidable, for applications where these conductors function by coupling with the power lines [18]. In PLatter, however, we leverage this as a benefit. Specifically, the PLatter reader measures the RF characteristic impedance of the power line by injecting a carrier signal into the power socket. The key enabler here is the terminating impedance mismatch of the power line due to the outlets in other rooms that are either left open or connected to appliances with mismatched impedance. This creates a reflected wave back to the reader. Meanwhile, each tag attached to the walls toggles between two internal impedance values (instead of high-power radio transmission), inducing recurrent patterns of parasitic impedance to the power line. This fluctuation is minute enough to ensure no harm to the normal operation of the power grid, but is readily detectable and decodable by the PLatter reader. Therefore, PLatter tags can function at much lower power because they do not need an active radio front-end. Instead, they only need an antenna with impedance switching capability to couple with the nearest power line.

The core challenge, however, is that the power line cables are designed to deliver AC power signals at 50/60 Hz and significantly suffer from impedance mismatch and signal attenuation in Radio Frequency ranges. A high impedance mismatch between the reader and its power line interface can result in a significantly high reflection coefficient, preventing the carrier signal from entering the grid and impacting communication. In addition, the characteristic impedance of power lines varies depending on cable length and geometry, which complicates the impedance matching circuitry even more. Further, the impedance at the reader interface may change over time as appliances are turned on/off or switch their operating states. This creates a standing wave inside the cables that varies with time, which significantly affects the performance of the backscatter network.

To overcome these challenges, we design an intelligent reader with adaptive frequency and impedance tuning capabilities to actively maintain tag detection. As such, the reader constantly monitors the input impedance of the cable and will accordingly tune the injected car-

rier frequency or the on-board impedance matching network to discover tags. In addition, PLatter also adapts to new appliances connecting to terminal outlets. The key intuition is that any change in the power line network causes a spike in the characteristic impedance, which is detectable by the reader and can be adapted to.

The second challenge is the design of an ultra-low-power tag that can create detectable parasitic impedance changes in the power line infrastructure. For this, we design a backscatter tag that switches between selected load impedance values to induce a distinct modulation pattern on the parasitic impedance sensed by the reader. We further improve tag detection by applying MAC-layer coding (e.g., PN codes) specifically for larger power line networks (e.g., in a warehouse). This also enables multi-tag detection by leveraging orthogonal codes per tag. Finally, in support of an ultra-low-power tag architecture, we design PLatter as a unidirectional network where data only flows from the tags to the reader. This greatly simplifies the tag circuitry by not requiring any envelope detector, digital signal processing, or decoding module. We show that this design choice reduces the tag power consumption to as low as 5 μW, with which a tag can operate for 12.9 years on a coin-cell CR2032 battery.

We show the feasibility of PLatter with custom backscatter tags at 13.56 MHz, which are compliant to FCC regulations and safety measures, and two USRP N210 software-defined radios emulating a mono-static full-duplex reader with custom PCB front-end that enables dynamic impedance tuning and notch filters to safely connect the reader to active power lines. We deployed PLatter in an industrial environment with more than 952 m^2 floor space, along with up to 300 feet (91 meters) power cables in various geometries (Sec. 8.2), in non-line-of-sight (NLoS) and dynamic scenarios (Sec. 8.4), and with active power (Sec. 8.6). We show that PLatter enables ultra-low-power backscatter communication that achieves up to 4 Mbps data rate while only consuming 5 μW power at the tag.

Contributions: Our core technical contributions are:

- A building-scale backscatter communication system leveraging the power line infrastructure to achieve up to 4 Mbps data rate over 300 feet power cables.
- A novel parasitic impedance modulation scheme by varying the parasitic impedance that a PLatter tag induces on power cables through near-field coupling.
- A detailed evaluation of an intelligent reader architecture with dynamic impedance tuning and frequency selection capabilities for power cables with arbitrary shapes and connected appliances.

Limitations: In this paper, we focus on extensively evaluating the feasibility of backscatter communication through the power line system. However, there are many

---

[1]PLatter: Power Line Backscatter

different factors that can affect the performance of the system, which require extensive follow-up exploration. For example, while PLatter tags can be read through long cables, the wireless medium between the tag and the surface of the cable is still limited to a few tens of centimeters, limiting the tag placement only on walls and ceiling along the power cables. Sec. 9 further elaborates on the limitations and future research directions.

2 Background and Related Work

While the power lines inside homes and buildings are primarily designed to carry high voltage 50/60 Hz AC signals for power distribution, they can also be used to communicate data at higher frequencies by acting as a transmission line, a transmitting antenna, or a receiver. According to basic electromagnetic theory, a time-varying current in a wire will produce an associated time-varying electromagnetic field around the wire. Since the power lines in a building are essentially a collection of wires, they can be potentially used as antennas. Using power lines as RF antennas has been explored in various contexts since the 1920s [17]. Several works describe various forms of a line cord antenna [25, 46, 44], whereby a receiver is coupled to the power line to receive high-powered broadcasts from TV or radio stations. Power lines have also been examined as transmitting antennas to either distribute AM radio broadcast signals over the main power distribution grid, known as carrier current [11], or as intentional radiators for cordless phone system transmitter or in-home video distribution [37].

This paper explores the feasibility of enabling long-range ultra-low-power backscatter communication using power line infrastructure by measuring the parasitic impedance induced by nearby backscatter tags. The rest of this section elaborates on other related work in both the power line and backscatter contexts.

Power Line as a Transmission Line: In a Power Line Communication (PLC) network, both transmitters and receivers are connected directly to the power line and communicate their data directly over the line. This has been widely used in home automation tasks, leading to protocols such as X10 [3], Insteon [1], and HomePlug [45]. Smart metering is currently a leading application for these systems. Today, high data rate PLC is a commercial reality known as broadband over power lines (BLP), and BLP modems can be purchased for various home or office applications with OFDM PHY layer and CSMA/CA MAC layer protocols.

Power Line as a Transmitting Antenna: Carrier current [11] is a popular method from the 1970s that uses the power lines as transmitting antennas for low-power AM broadcasts. A carrier current system can cover an entire building or even a group of buildings at low transmission power, which makes it ideal for localized radio such as college and high school radio stations. Power Line Positioning (PLP) is another technology that uses power lines in a building to track the location of small sensors throughout the home [36, 32], or detect the presence of objects, people, and their activities [20, 29, 30]. Both of these technologies rely on lower radio frequencies (e.g., 300 kHz to 20 MHz) for best performance. Similarly, leaky feeder systems [26] use a coax cable running along the tunnels, underground mines, or railways for emitting and receiving radio waves, functioning as extended antennas. However, the cable is specifically designed for radiating, with slots cut into the outer shielding. PLatter purely relies on existing power line infrastructure in any building to read backscatter tags.

Power Line as a Receiving Antenna: Early research on power line position systems demonstrated that both AM and VHF FM radio broadcasts can be also heard and demodulated by the power line as a receiving antenna [13]. SNUPI [10] and more recent follow-up work [37] have also shown a uni-directional communication network from wireless sensors to base stations attached to home power lines. However, the sensors are still actively transmitting high-power wireless signals, which are then sensed by the receiver attached to the power line. In contrast, PLatter allows the tags to be ultra-low-power by eliminating the need for an active radio front-end.

Wireless Backscatter Communication: Traditional backscatter networks such as RFID [43] and NFC [12] rely on energy harvesting from a carrier wave to power battery-free tags, which then send sensory data to the reader. However, a majority of these networks are limited to either a short range (< 10 m) or a low data rate. Recent work on LoRa [39, 23] and WiFi backscatter [7, 47] target these challenges by either using LoRa-compliant chirp signals to extend the range, or more complex modulation techniques such as OFDM to improve the data rate. However, some of these backscatter systems still assume a separate power source in the near proximity of the tags or are limited to line-of-sight scenarios to achieve long-range communication. NetScatter [23] is the closest wireless backscatter network that provides multi-room coverage by using chirp spread spectrum coding, but at the exchange of a reduced data rate of 100-150 kbps.

In contrast to this rich prior work, PLatter explores the feasibility of a building-scale backscatter communication with up to 4 Mbps data rate using the power line infrastructure already available in every building.

FCC Rules and Regulations: Power line communication and carrier-current systems are generally considered as "Restricted Radiation Devices" under Part 15 of volume 11 in FCC rules and regulations, which specifies the
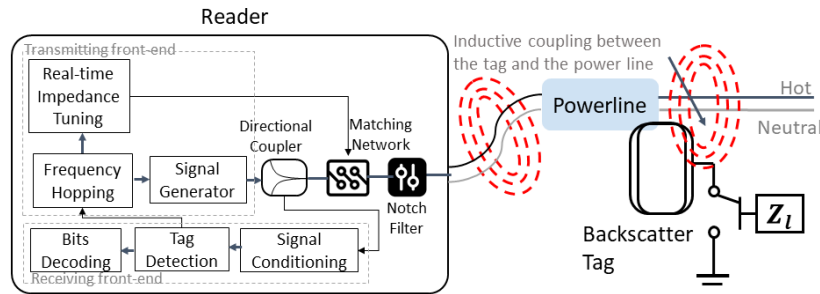
Figure 2: PLatter Overview

maximum electric field strength that an EM radiator is allowed to emit. To comply with these regulations, PLatter is specifically designed as unidirectional and is only relying on the induced parasitic impedance; hence the tags are not designed to emit wireless radiation. In addition, the EM field strength meter shows compliant readings around the cables as PLatter reader injects carrier signals. Our current implementation exploits the unlicensed ISM band at 13.56 MHz, which also makes PLatter compatible with NFC systems.

3   PLatter Design

In contrast to typical backscatter systems, PLatter leverages the existing power line infrastructure to increase the communication range between ultra-low-power tags and a single reader. Attached to an outlet, the reader continuously measures the characteristic impedance of the power line and looks for variations in parasitic impedance to detect and decode the tags' data. To minimize power consumption and network complexity, PLatter is designed to be unidirectional from the tags to the reader for upstreaming sensory data to the reader. Therefore, the tags perform modulation whenever sensory data needs to be sent, independent from the reader operation. Meanwhile, the reader performs real-time impedance tuning and frequency adjustment to adapt to network changes. Fig. 2 shows an overview of PLatter.

Designing Reader's Transmission: The reader's signal is not only subject to attenuation and noise, but also susceptible to appliances being turned on/off. In addition, it experiences frequency-selective fading due to the standing waves. To mitigate these, PLatter adopts an adaptive reader design that continuously monitors network changes by measuring the input impedance. It hops towards a favorable frequency if the current carrier signal is heavily attenuated, or performs real-time impedance tuning with an impedance matching network at its interface to the power line network. Sec. 4 elaborates our design.

Tag Design and Data Decoding: Sec. 5 details the tag hardware, its modulation scheme, and a decoding pipeline. PLatter's modulation scheme leverages the fact that electromagnetic fields of high-frequency injected signals into power lines couple with other nearby conductors. Hence, as long as the carrier signal traverses the power line (reader's task in Sec. 4), one can enable long-range backscatter via the power line by coupling with ultra-low-power tags along the power line. Sec. 5 elaborates this modulation scheme and further shows how PLatter achieves robust and efficient tag detection and decoding with low-cost and low-power tag circuitry.

4   Designing Reader's Transmission

In this section, we first describe PLatter's power line backscatter channel model (Sec. 4.1) and the choice of frequency band of operation (Sec. 4.2). We then detail two key reader designs: (1) adaptive frequency hopping (Sec. 4.3); (2) real-time impedance tuning (Sec. 4.4).

4.1   Power Line Backscatter Model

In PLatter, the carrier signal from the reader propagates through electric wires and various discrete components such as transformers. Part of the signal attenuates, while the remaining energy gets reflected in the case of impedance mismatch at the termination. The end result is a standing wave that operates on the wire. This wave is further modulated due to the presence of backscatter tags as it switches between different impedance values, modulating the wiring system's overall impedance.

Power Line Channels are Frequency-selective: A natural property of the standing wave created on the wire is that it has several nulls whose locations are dependent on frequency. To see this in practice, Fig. 3 illustrates the attenuation in typical NM-B 14/2 power cables of different lengths (25, 50, and 100 feet). A single tone (0-1 GHz) is injected into the cable and the reflection characteristic (S11) is measured while the other end of the cable is left open for minimal terminating loss. We see that the attenuation increases with both frequency and cable length. In addition, the non-linear behavior of the cable at certain frequencies is due to impedance mismatch, which leads to standing waves, or high reflection at the entrance of the cable.
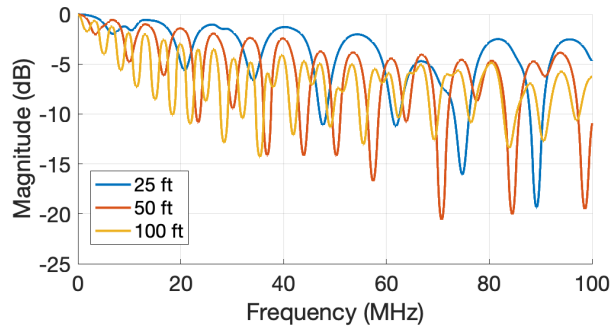
Figure 3: Reflection characteristics (S11) of cables with different lengths demonstrate the standing wave effect and attenuation trend as frequency and cable length vary.

Time-Varying Channels: The power line channel has been widely studied. Much like a traditional wireless channel, it also exhibits multipath effects due to the superposition of various signal paths. We refer the reader to [27] for a detailed channel model. Two points are worth noting: (1) appliances can greatly influence the power line channel both by injecting high-frequency noise and changing the system's overall impedance; (2) some elements, such as a charged transformer, may cause periodic time-variations in power line channels [28]. These, if any, fall around the order of *ms* and can simply be avoided by modulating signals of the order of μs.

Why can the reader sense tags' modulation? To understand how the tag modulation is detectable, we refer to Ampere's law [22], where a magnetic field is generated by a group of closely bundled wires and the current flowing through them. For the power line infrastructure, the hot and neutral wires carry currents in the opposite direction, canceling the magnetic field generated by the 50/60 Hz AC signal. It is, however, possible to create an electromagnetic field when injecting a higher frequency signal (e.g., for the purpose of impedance measurements). In this case, the power line can be crudely visualized as a gigantic coil, causing near-field inductive coupling with a secondary coil (e.g., the backscatter tag), as shown in Fig. 2. For modulation, the tag alternates the load on its coil and creates a varying parasitic impedance that can be detected by the reader (more details in Sec. 5).

4.2 Choosing Frequency of Operation

Given the mono-static setup of PLatter's reader, we first study the reflection behavior of multiple power lines across a wide range of frequencies between 0-100 MHz, shown in Fig. 3. While we observe a gradual increase in the amount of signal attenuation with length and frequency, the attenuation at 10-20 MHz is comparatively small across different lengths of cables, with reasonable sizes of (coupling) antennas. This allows the reader to receive a reflected signal for the purpose of computing
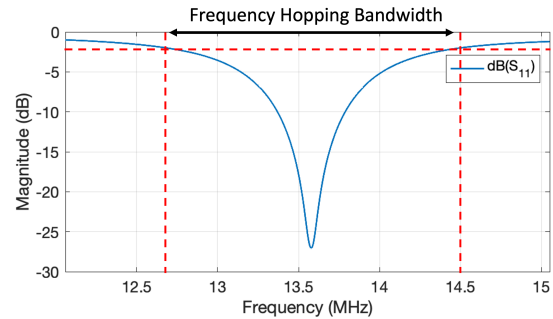

Figure 4: A typical NFC antenna resonates at center frequency 13.56 MHz and minimum of 50% antenna delivery for a ~ 2 MHz bandwidth.

impedance. Among the frequencies below 20 MHz, we choose to design PLatter in the unlicensed ISM band of 13.56 MHz. This makes PLatter inherently compatible with NFC in terms of tag antenna design.

To select the operating bandwidth, we need the tag antenna to resonate well over the entire bandwidth, so that it can effectively induce the desired parasitic impedance variations for data transfer. To examine this, we select an off-the-shelf NFC antenna [2] with a center frequency of 13.56 MHz. As shown in Fig. 4, the antenna has a highly narrow beam, but we can still expect about 50% of delivered power on frequencies between 12.5 MHz and 14.5 MHz (i.e., reflection powers below -2.92 db). Therefore, we select this bandwidth for frequency hopping with potential steps of every 500 kHz.

4.3 PLatter's Frequency Hopping Design

The core challenge in designing PLatter is the potential standing wave effects due to the impedance mismatch between the power line and the reader, which create deep nulls at certain positions along the cable. The key intuition that PLatter leverages is the frequency-selective behavior of the power line. Specifically, while one frequency can cause a deep null at certain location along the cable, the effect could be completely reversed at a slightly different frequency. An example of this is shown in Fig. 5, where we see more than 5 dB improvement in the tag SNR by slightly shifting the frequency of the carrier signal. However, it is critical to find the best carrier frequency quickly for high data rate communication. Thus, PLatter leverages the continuous and locally convex behavior of the power line across frequencies (as in Fig. 5) and defines a Stochastic Hill-Climbing algorithm with random initial points. At every iteration, the reader measures the SNR of the reflected signal and searches for the tag reflection (explained in Sec. 5.3). It then adjusts the carrier frequency and continues this operation until no improvement on the tag SNR can be found.

Another requirement of this frequency hopping algorithm is a mechanism to quickly and reliably detect
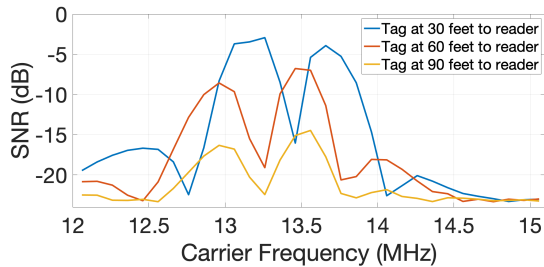
Figure 5: PLatter leverages the frequency-selective behavior of the power lines to improve the tag SNR by intelligently shifting the frequency of the carrier signal.

whether there is an active tag present. To achieve this, we design each tag to perform a known modulation in the form of a physical layer preamble before sending data. This preamble is defined as a sequence of periodically switching between two impedance values for a fixed time, which appears as a square wave with a switching frequency of $f_p = 1/T_p$. This can be easily detected in the frequency domain with a simple FFT. This way, the reader can quickly hop between frequencies and search for active tags, then fix on a frequency to receive data.

4.4 Real-time Impedance Tuning

One of the essential requirements of PLatter to work is an impedance matching network between the reader and the power line so that the carrier signal can enter the power line network and get reflected back (at other ends of the cable; e.g., an open outlet). While different matching network architectures are proposed for traditional Power Line Communication (PLC) systems [15, 38, 40], PLatter's backscatter setup necessitates a different architecture. In a completely wired setup like PLC, where both the transmitter and receiver are connected to the power line, impedance matching is required at both of them, incurring much higher noise [44]. PLatter, instead, relies on a mono-static reader setup in which the matching network is shared between the transmitting and receiving radio chains of the reader (Fig. 2). However, the time-varying channel conditions (Sec. 4.1) remain a challenge in designing such a matching network. In addition, a new scalability challenge arises as the characteristic impedance of the power lines in different buildings may be drastically different depending on the geometry and layout of the building.

PLatter addresses these challenges by performing real-time impedance measurement and tuning. It continuously monitors network changes due to appliances being turned on/off and accordingly adjusts the matching network. The key enabler here is that PLatter does not necessarily require a perfect matching, since it only relies on the variations of parasitic impedance to decode tags' data. Hence, approximate impedance matching is
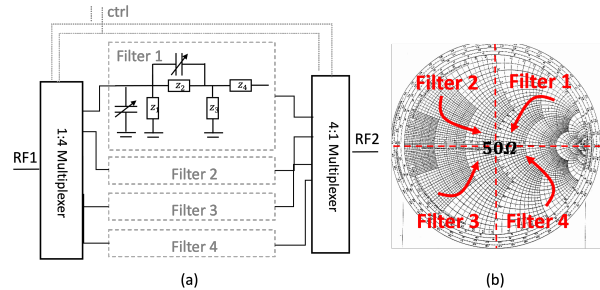

Figure 6: PLatter's modular matching network design that provides real-time impedance tuning by inferring channel conditions and adapting accordingly.

sufficient as long as the reader obtains sufficient reflected signal power. This greatly reduces the technical difficulty of PLatter – it would be much easier to design a tunable matching network that targets approximate rather than precise tuning to 50 Ω for example.

As such, we design a tunable matching network (Fig. 6 (a)) with four sets of analog filters, each constructed as a series of two cascaded L networks. These filters can be selectively populated to form different circuit structures (e.g., L-shape or π-shape). In addition, they also include programmable and digitally tunable capacitors. The circuit structures and corresponding components (R, L, or C values) are carefully selected such that each network can cover roughly a quadrant of the input impedance viewed in the Smith Chart (Fig. 6 (b)). With this, PLatter can coarsely match any input impedance encountered in our experiments, which enables tag detection and decoding.

5 Tag Design and Data Decoding

In this section, we describe PLatter tag's data modulation scheme (Sec. 5.1), its hardware (Sec. 5.2), the corresponding detection and decoding pipeline (Sec. 5.3), and scalability to multiple tags (Sec. 5.4).

5.1 Tag Data Modulation

Similar to other inductive coupling based backscatter tags (e.g., NFC), our primary design requirement for PLatter's tag is to ensure sufficient coupling with the power line when they are in close proximity. The tag must then modulate its digital signals onto power lines. On one hand, frequency modulation (FM), seen in many traditional PLC deployments, provides enhanced robustness but the data rate is relatively low (e.g., tens of kbps at most). On the other hand, phase modulation (PM) or amplitude modulation (AM), commonly used in backscatter systems, are easier and cheaper to implement but are less robust. PLatter's design choice is to perform what we call *Parasitic Impedance Amplitude Modulation* as a variation of Amplitude Shift Keying (ASK). The
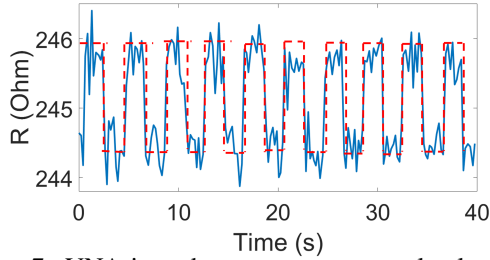
Figure 7: VNA impedance measurement clearly shows the tag placed at 8 cm away from the middle of a 100-ft cable modulating the parasitic impedance.

power of parasitic impedance is a function of the tag's reflection coefficient as

$$\Gamma = \frac{Z_T + Z*_A}{Z_A - Z_T} \tag{1}$$

where Z_A is the power line characteristic impedance and Z_T is the impedance of the tag terminal. To achieve the highest variation in parasitic impedance, we choose to switch between short and open with expected nominal 0 Ω and infinity impedance values.

As a preliminary study, we deployed a 100-ft cable, one end terminated with a Vector Network Analyzer (VNA) and the other left open (SMA open cap). We placed a tag at the middle (i.e., 50 feet away from the VNA) with a distance of 8 cm to the cable. The tag is set to switch between short and open loads with a very slow rate (every 2 sec). As shown in Fig. 7, the impedance measurements clearly capture the tag's modulation. Yet, we should also note the small scale of changes (i.e., parasitic impedance) compared to the absolute value of the cable's characteristic impedance.

In addition, to improve the SNR that a tag experiences at different locations along the cable, PLatter also implements channel coding to effectively pull up the SNR. We choose traditional convolution coding due to its simplicity to implement and a wide range of coding gain-coding rate choices. The gain will be implicitly shown in Sec. 8 where we evaluate PLatter's data rate performance.

5.2 Tag Hardware Design

To achieve ultra-low power consumption at the tag with a high data rate, PLatter exploits a unique hardware design. First, we shift most of the system complexity to the reader with a minimal architecture at the tag. For example, with uni-directional communication from the tag to the reader, we do not need any envelop detector, decoding component, or synchronization module. It is entirely the reader's task to detect active tags and decode their data. Then, to achieve a high data rate, we leverage high-speed SPDT RF switches with nanosecond switching rate, controlled by a low-power microcontroller (Fig. 8 (a)). As such, the tag can easily support
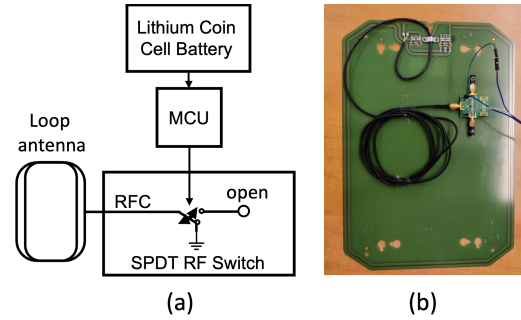

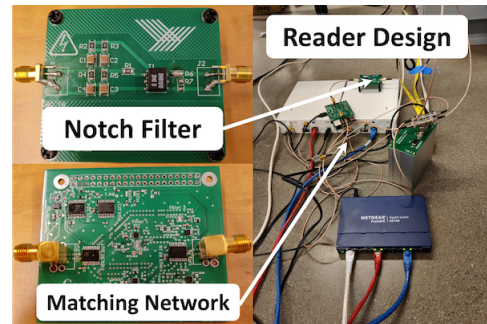Figure 8: Minimal architecture of a PLatter tag allows ultra-low power consumption.

Figure 9: PLatter's reader consists of a 60 Hz notch filter and a matching network that tunes impedance live.

orders of Mbps data rate with parasitic impedance amplitude modulation. Fig. 8 (b) shows its dev-kit prototype.

5.3 Tag Detection and Decoding

Next, we describe how the PLatter reader extracts the modulated parasitic impedance from the reflected signal. Our decoding algorithm works in two main steps: (1) signal conditioning to remove sudden variations and noises in the measurements; (2) decoding the backscattered bits.

Signal Conditioning: The goal is to remove high-frequency temporal variations in the measurements due to background noise or sudden impedance changes caused by connected appliances. We measure a moving average from the channel measurements that is defined based on the upper bound of the networks' data rate. In addition, if the measurement contains colored noise (as seen in Sec. 8.6), PLatter adopts further denoising.
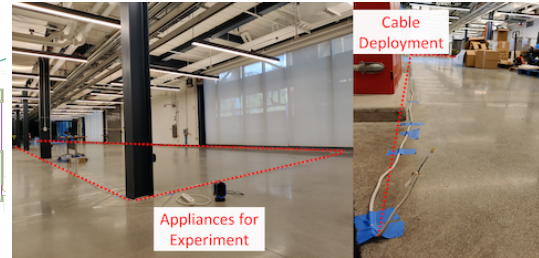
Decoding Bits: PLatter applies simple thresholding on the output of the previous step. Specifically, if the channel measurement is above the threshold, the backscattered bit is considered as a "1", and a "0" otherwise.

5.4 Scaling to Multiple Tags

In PLatter's design, multiple tags may send data to the reader simultaneously. While many existing medium access control protocols can be implemented for PLatter's
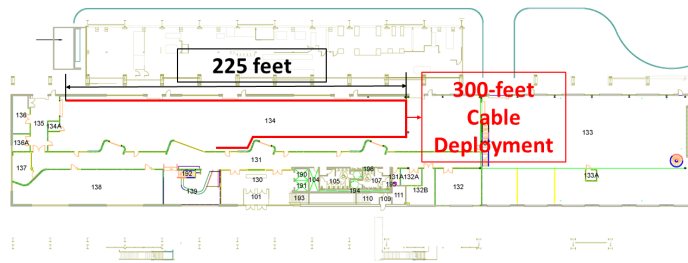
(a). Deployment layout.   (b). Testbed photos.

Figure 10: PLatter is evaluated in an industrial environment with up to 300 feet power cables.

power line backscatter network (e.g., ALOHA, TDMA, or CSMA/CD), we leverage PN code assigned to each tag. This enables concurrent communication while keeping the tag circuitry simple and ultra-low-power. In addition, it eliminates the need for a bi-directional link and any sort of synchronization. To achieve this, PLatter incorporates a shift register holding a PN code before the RF front-end. The code length corresponds to the maximum number of concurrent tags that the system needs to support, which is configured prior to deployment accordingly. As an example, a 63-bit PN code requires a 6-bit shift register and supports 63 concurrent tags.

6   Implementation

Reader Front-end: The PLatter reader, shown in Fig. 9, consists of two USRP N210s with BasicTX/BasicRX daughterboards, synchronized to the same clock. They are connected with a directional coupler to emulate a full-duplex reader, which is controlled by shell and C/C++ scripts running on an ASUS 8G RAM 64-bit laptop with Ubuntu 16.04. The reader adaptively selects a frequency between 12.56 MHz and 14.56 MHz and transmits a carrier tone. This signal first travels through the tunable matching network, which is controlled by the laptop and provides four candidate channels, then enters the power line network to capture tags' signal. The reflected signal from the power line first enters the 60 Hz notch filter, then gets captured by the reader with a sampling rate of 25 Msps and processed offline in MATLAB. The notch filter effectively removes active grid noises to guarantee a proper dynamic range of the received signal and protects the reader from severe damage.

Tag Hardware: The PLatter tag (Fig. 9) consists of a minimal hardware, in which an antenna is connected to a 3-port HMC284AMS8G SPDT RF switch [5]. The antenna is a two-loop coil fabricated on PCB and tuned to a center frequency of 13.56 MHz. The other two ports of the RF switch are terminated with short and open SMA caps, resp. The switch is then controlled with either Raspberry Pi for benchmark experiments or MSP430FR5994 MCU [4] for power analysis. The entire

tag circuitry is designed in favor of low cost and power consumption, with a nominal 50 Ω impedance and all the required impedance matching shifted to the reader side.

Tag Power Consumption: One of our key design challenges was to minimize tag energy consumption. We pair the RF switch with a MSP430FR5994 MCU. In the active state of transmitting information, PLatter uses 4.95 μW of power; otherwise, the MCU remains in an ultra-low-power sleeping mode (LPM4) (1.05 μW) with an internal low-power, low-frequency oscillator running. Assuming the tag sends 100 packets per day, 20 bytes each at a speed of 1 Mbps, we achieve a daily expenditure of 403.7 mJ. If paired with a small form-factor 3V CR2032 lithium coin cell (235 mAh), we predict that a tag could offer operation for 12.9 years, assuming an efficiency of 75% and no battery self-discharge.

7   Evaluation

Experimental Setup: We deployed multiple NM-B 14/2 cables with different lengths between 25 and 300 feet in an industrial warehouse (formerly, a steel mill) designed to serve as a smart manufacturing testbed (10250 sq. ft.). Fig. 10(a) shows the cable layout, and Fig. 10(b) is taken in the experimental space. Both ends of the cables have SMA connectors soldered for easier connectivity. For safety and controllability, in most of our experiments, the cables do not carry active AC power (i.e., static), except in our active power test (Sec. 8.6), where we instead use the building's existing power grid. Yet, we always have the 60 Hz notch filter in the circuit to ensure consistency.

Evaluation Metrics: We examine and report two main performance metrics: (1) SNR in dB, which reflects the signal strength a tag can enjoy at a certain location; it does not account for any gains from coding and software denoising. (2) Achievable data rate in bit-per-second (bps), which generally coincides with the trend of SNR while implicitly including gains from coding and denoising. Mean values across experiments are reported and error bars denote standard deviation.

Baselines: We do not depict the comparison between PLatter and an active near-field (NFC-based) over-the-

(a). Tag at the start of a cable. (b). Tag at the middle of a cable. (c). Tag at the end of a cable.
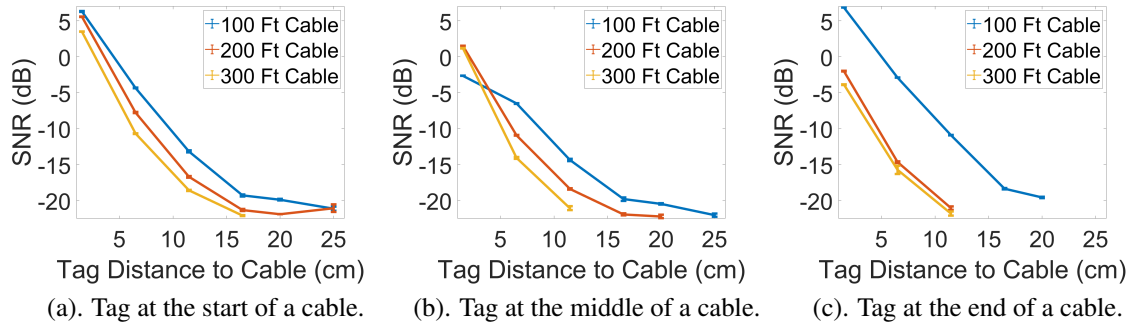
Figure 11: PLatter's SNR performance when a PLatter tag is placed at the (a) beginning (b) middle (c) end of cables of different lengths (100 feet, 200 feet, and 300 feet).

air transmission system with an equivalent power given the very short range of NFC tags (a few cms). We note that a far-field equivalent at 13.56 MHz would require a huge antenna (the wavelength is 22 m) that is impractical for an indoor space. However, we do evaluate the effectiveness of PLatter's matching circuit in Sec. 8.3 and we show PLatter achieves a maximum data rate comparable with other backscatter systems.

8 Results

We perform a thorough evaluation of PLatter where we examine multiple factors that impact PLatter's performance – cable length and geometry, tag position, electrical appliances connected, separating material between a tag and the cable, and active power.

8.1 Cable Length and Tag Position

Method: In this section, we evaluate PLatter's SNR and data rate with a single PLatter tag, and we vary three system variables: (1) total cable length; (2) tag's position w.r.t. the reader, i.e., the cable length between the tag and the start of the cable; (3) tag's distance to the cable, i.e., the closest distance between the tag and the cable along the cable's normal direction. The cable is always terminated with the reader at one end and an SMA open cap at the other, emulating an open power outlet.

Result: Fig. 11 shows PLatter's SNR performance at three different tag positions along the cable (beginning, middle, and towards the end). We see that as the tag moves farther away from the cable, the SNR decreases due to weaker inductive coupling; the SNR also drops as the total cable length increases due to higher signal attenuation inside the cable and weaker reflection received by the reader. Comparing across multiple figures, we observe a higher SNR when the tag is placed close to the beginning of the cable, since the carrier signal gets modulated before it experiences severe attenuation. Based on our observation: (1) a tag can be detected when SNR
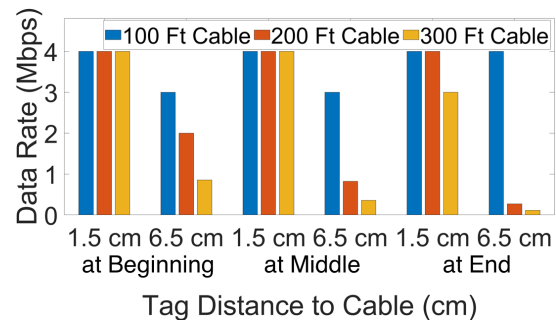

Figure 12: PLatter's end-to-end data rate performance when a PLatter tag is placed at the beginning, middle, and end of cables with different distances to the cables.

is as low as -21 dB; (2) a tag can be detected and decoded between -16.5 dB and -21 dB but it suffers from a low data rate; (3) a tag's maximum data rate is around 1 Mbps when SNR is around -9.7 dB and increases to 4 Mbps when the SNR is as high as 5 dB.

Fig. 12 shows PLatter's data rate performance with 6 different configurations – 2 tag distances (1.5 and 6.5 cm) and 3 tag positions (beginning, middle, and end). The overall trend closely follows Fig. 11. Specifically, a tag at a favorable location can modulate at the maximum data rate; otherwise, it adds more redundancy to its data and paces down. In our experiments, an SNR lower than -16.5 dB leads to zero data rate because the tag signal can no longer be decoded; yet, there is still a chance for the reader to detect its presence. Overall, the maximum data rate PLatter can provide is 4 Mbps; this is comparable to the best backscatter-based state-of-the-art [7], yet achieved with a reader farther away from the tag (300 ft) by reusing power lines as a communication medium.

From Sec. 8.2 on, we fix the total cable length to be 100 ft and choose a representative subset of tag locations to better demonstrate the influences from other factors.

8.2 Cable Geometry

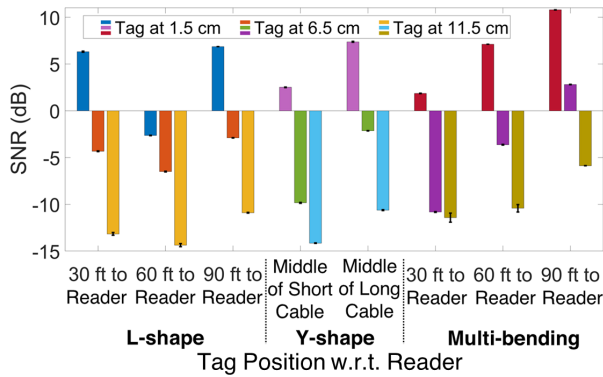**Method:** In this section, we examine three different cable geometries with a total length of 100 ft (two 25-ft ca-

Figure 13: The tag SNR varies as the cable geometry changes, yet PLatter can operate normally in all cases.
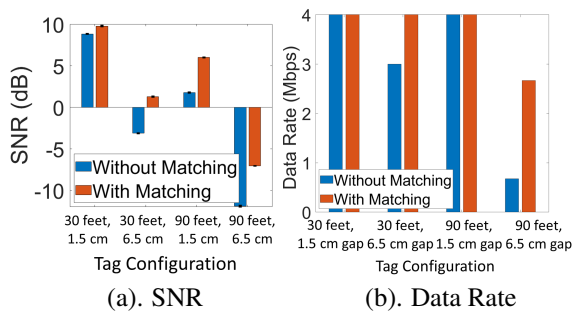

(a). SNR (b). Data Rate

Figure 14: Effectiveness of PLatter's Matching Circuit

Figure 15: The tag SNR varies slightly as an appliance's internal circuit changes, and PLatter can adapt coordingly with its matching network.

Figure 16: PLatter with different materials.

bles and one 50-ft cable). Specifically, we consider three configurations: (1) L-shape connection forms a long 100-ft cable with a corner point at the middle (50 ft); (2) Y-shape connection has one splitter to form two branches (a 50-ft cable (long) connected with two 25-ft (short) branches); (3) Multi-bending connection has a number of bending points along the cable.

Result: Fig. 13 shows PLatter's SNR performance under different cable geometries at representative locations. Despite minor variations in SNR, we verify that PLatter continues to operate under different practical cable geometries, including at the branches of the Y shape.

8.3 Impact of Electrical Appliances

Method: In this section, we examine both the effectiveness of our matching network and the impact of electrical appliances. We choose two multi-state appliances: (1) a desktop heater (on/off) as a representative of appliances that turn electrical power to heat; (2) a surge protector (on/off) commonly used in daily life. They change the overall impedance of the power line, and PLatter is expected to adaptively tune its matching network. To connect an appliance to our cable, we use a custom-designed SMA-to-plug converter. We use a single 100-ft cable and choose a representative subset of tag locations. Note that only in this experiment, we include SNR values that were
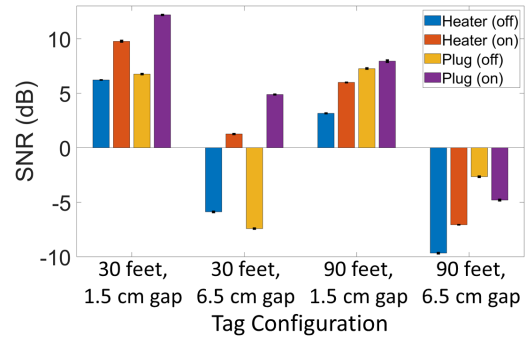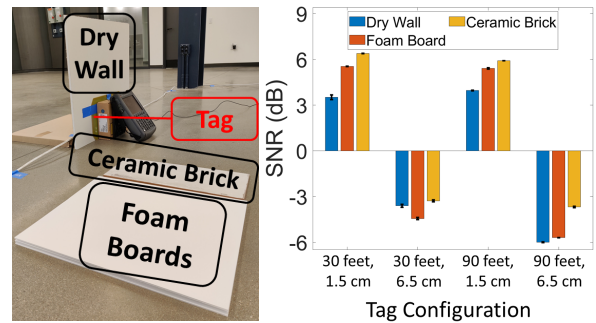
measured when PLatter's matching network was off.

Result: We show the effectiveness of the matching network in Fig. 14 by attaching a heater (on) to the cable. We see an overall SNR improvement across different tag configurations, though the actual amount may vary. The impact of appliances is shown in Fig. 15. The reported numbers have included gains from the matching network. In general, for a certain tag configuration, the SNR tends to be higher when the appliance is on (i.e., its internal circuit is connected); even when the appliance is off, with the matching network PLatter still manages to maintain a reasonable SNR and hence data rate.

8.4 Influence of Separating Material

Method: We evaluate the impact of different wall materials between the tag and the power line. Specifically, we consider 3 common materials: dry wall, foam board, and ceramic brick (Fig. 16), with their thickness of around 1.3 cm (1/2 inch). Again, we use a single 100-ft cable and stick to a representative subset of tag locations.

Result: Fig. 16 shows that PLatter's performance slightly varies as we change the separating material; in general, the dry wall panel tends to introduce more attenuation, but PLatter can still maintain a favorable SNR that is way higher than the decoding threshold.
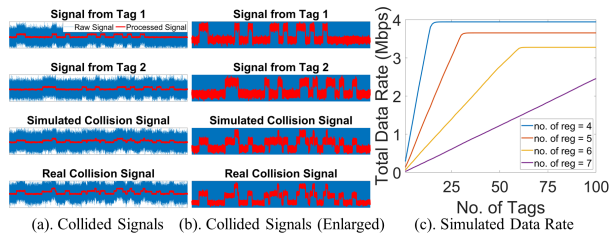
(a). Collided Signals   (b). Collided Signals (Enlarged)   (c). Simulated Data Rate

Figure 17: PLatter uses a simulation route to examine the collective data rate in a multi-tag scenario.

8.5 Multiple Tags

Method: In this section, we explore PLatter's performance with multiple tags. First, we verify that signals from multiple tags add up as expected by placing two tags along a 100-ft cable at 30 feet and 90 feet, respectively, with a distance of 1.5 cm to the cable. The tags periodically transmit a specific bit sequence and the reader receives their colliding signal for further analysis. Next, to evaluate the scalability of PLatter in multi-tag scenarios, we conduct a trace-driven simulation of collisions from 1 to 150 tags. We use the simulation route to carefully and exhaustively model various relative timing offsets between tag transmissions to study their overall impact. Our study is informed by actual signals collected from 3 tags placed at 30, 60, and 90 feet from the reader, and for each position, we include both 1.5 and 6.5 cm as their potential distance to the cable. We then engineer a collision in software under different timing conditions and calculate the total data rate across all tags.

Result: Fig. 17(a) and (b) show that signals from multiple tags add up with each other as expected. Even though the raw received signals (shown in blue) are quite noisy due to background noise, PLatter can still successfully detect the tags by correlating each tag's PN-code. In principle, PLatter only relies on the variations of parasitic impedance, not absolute impedance values. Fig. 17(c) shows the simulated data rate for multiple PLatter tags. It should be noted that the length of PN code and shift register at each tag is defined by the network size (i.e., the maximum number of concurrent tags supported by the system). This imposes a trade-off between the maximum collective data rate and the number of tags, where the maximum total data rate drops down slightly as the number of registers increases.

8.6 Response on Active Power Lines

Method: In this section, we evaluate PLatter with an active power grid in an industrial environment. We connect one end of a 25-ft cable to our reader, with the 60 Hz notch filter and matching circuit in series, and plug the other end into a surge protector, which is further connected to the active power outlet on the wall of the build-
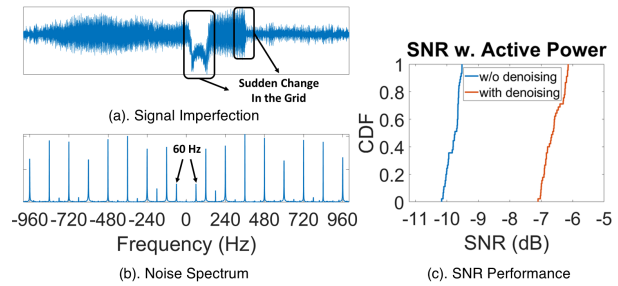

(a). Signal Imperfection

(b). Noise Spectrum   (c). SNR Performance

Figure 18: Both sudden signal imperfections and periodic harmonics have been observed in the active grid; PLatter's corresponding denoising technique greatly helps to improve the SNR.

ing. This forms a large active power grid with an estimated total length of more than 500 m; the building was operating normally with a variety of appliances running on the power grid. The reader is powered by a portable battery pack and protected by another surge protector. A single PLatter tag is put at three different positions along the 25-ft cable with a fixed distance of 1.5 cm to the cable, and we carry out 15 independent measurements for each case, creating a total of 45 experiments.

Note that the cable geometry here is completely different from those in Sec. 8.1-Sec. 8.5; meanwhile, it was infeasible to shut down the whole grid to collect static experimental data in this building. Hence, static test results (when PLatter is connected to the whole building's grid but without active power) are not included here.

Result: We first analyze the noise introduced by the active grid. Fig. 18(a) shows a representative trace of the signal in time domain with an obvious imperfection in the middle. This kind of sudden noise has been observed multiple times throughout the experiment and it does not have a fixed pattern. Meanwhile, we have also observed periodic noise components resulting from the 60 Hz AC signal. Fig. 18(b) shows the frequency spectrum of the periodic noise components. The first pair of peaks correspond to the 60 Hz bin, and we see multiple higher-order harmonics. While our notch filter significantly suppresses 60 Hz and its neighboring frequencies so that they would not damage the reader, the residual harmonic noise is still large enough to be detected by the reader. To deal with both kinds of noise, PLatter denoises in software processing to pull up the SNR. Note that this specific denoising is not present in Sec. 8.1-Sec. 8.5.

Fig. 18(c) shows the SNR observed in multiple trials along the cable. The tag was placed at different positions but its distance to the cable was fixed (1.5 cm). In this CDF plot, we see that in average PLatter's denoising technique brings 3-4 dB boost in SNR, although the actual improvement varies slightly along the cable. This makes it easier for the PLatter reader to decode the tag signal and has the potential to be further improved

by using more advanced notch filters proposed in the power line communication literature [33]. We admit that the SNR after boosting is still not within the most ideal range; yet, this can be mitigated with channel coding where we add redundancy into the transmitted data (as mentioned in Sec. 5). The experiment shows a positive sign of implementing PLatter in real-world settings with complex power grid conditions, albeit with lower data rate and range.

9 Discussion and Limitation

While PLatter takes a big step toward enabling building-scale backscatter communication using power line systems, there are still some open questions and a few avenues for future improvements.

Tag Proximity to Cables: PLatter requires the tag to be sufficiently proximate to the cables behind walls (within a few tens of cms – Sec. 8.1). While this may restrict the locations where PLatter tags can be deployed, such a solution is still desirable in intrinsically safe environments by just lying a passive cable in the environment for sensor communications. In addition, this requires the knowledge of power line locations behind the wall, which can be addressed by using stud finders during the sensor deployment. In our future work, we plan to study other modulation mechanisms at the tag to improve the tag detection rate at larger distances from the cable.

Variability in Performance: Based on our observation, PLatter's performance and range are highly dependent on several factors, specifically cable geometry, tag position (w.r.t. reader), distance to the cables, and active power variations. While PLatter can automatically adapt to these changes, it may experience performance drops and failure in highly dynamic setups.

Uni-directional Communication: PLatter in its present form is uni-directional from the tag to the reader. We believe that in principle, modulating information in the reverse direction while keeping low-cost and low-power is possible, and we leave this for future work.

Tag Scalability: While PLatter provides a proof of concept for multi-tag backscatter communication, the overall data rate of the network can be enhanced using MAC protocols such as TDMA or slotted ALOHA with minimum collisions. New backscatter MAC protocols like NetScatter [23], with power line time synchronization mechanisms [34] enabled, can further improve PLatter's overall performance; integrating them with PLatter is one interesting part of our future work.

Impact of Parasitic Impedance on Connected Appliances: Although large parasitic impedance can be generally problematic at high frequencies since it might sharply change voltages or currents, the amount of parasitic impedance induced by PLatter tags is negligible as the tag is completely passive in a sense that it has no active radio front-end and hence no radiation.

Electrical Wiring Complexity: In addition to controlled experiments, PLatter has been evaluated in an industrial manufacturing building, where the power grid was operating actively (Sec. 8.6), which provides the feasibility of backscatter communication using power line systems. However, we acknowledge that electrical wiring in some buildings can be complex with various hardware components in the line, which has long been a challenge in power line communication research.

10 Conclusion

This paper presents PLatter, a system that allows ultra-low-power backscatter tags attached to walls to communicate with a reader several hundreds of feet away. PLatter achieves this by using the existing power lines behind the walls as a communication medium, without physically being connected to them. To achieve this, PLatter modulates parasitic impedance on the power line system, which allows data to be recovered at a high rate at a single centralized reader through a large indoor facility. We present a detailed proof-of-concept evaluation of PLatter, exploring its strengths and weaknesses in a large indoor industrial testbed. While this paper broadly explores the concept of power line backscatter, our future work hopes to stress test the system at scale in diverse environments and explore higher-layer protocol designs.

Acknowledgement

References

[1] Insteon. `http://www.insteon.net/`.

[2] Nfc antenna 1663.000. `https://www.digikey.com/en/products/detail/feig-electronic/1663-000-00/1015926`.

[3] X10: Standard and extended x10 protocol. `http://software.x10.com/pub/manuals/xtdcode.pdf`.

[4] Texas Instruments MSP430FR5994 microcontroller. `https://www.ti.com/product/MSP430FR5994`, 2020.

[5] Hmc284ams8g, analog device, 2021.

[6] Kannan Srinivasan Athreya, Wei Sun, Bo Chen, and Vivek Sriram Yenamandra Guruvenkata. Mimo architecture for multi-user power line communication, December 29 2020. US Patent 10,879,959.

[7] Dinesh Bharadia, Kiran Raj Joshi, Manikanta Kotaru, and Sachin Katti. Backfi: High throughput wifi backscatter. *ACM SIGCOMM Computer Communication Review*, 45(4):283–296, 2015.

[8] Gabe Cohn, Daniel Morris, Shwetak Patel, and Desney Tan. Humantenna: Using the body as an antenna for real-time whole-body interaction. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '12, pages 1901–1910, New York, NY, USA, 2012. ACM.

[9] Gabe Cohn, Daniel Morris, Shwetak N. Patel, and Desney S. Tan. Your noise is my command: Sensing gestures using the body as an antenna. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, page 791–800, New York, NY, USA, 2011. Association for Computing Machinery.

[10] Gabe Cohn, Erich Stuntebeck, Jagdish Pandey, Brian Otis, Gregory D. Abowd, and Shwetak N. Patel. Snupi: Sensor nodes utilizing powerline infrastructure. In *Proceedings of the 12th ACM International Conference on Ubiquitous Computing*, UbiComp '10, page 159–168, New York, NY, USA, 2010. Association for Computing Machinery.

[11] Edwin H Colpitts and Otto B Blackwell. Carrier current telephony and telegraphy. *Transactions of the American Institute of Electrical Engineers*, 40:205–300, 1921.

[12] Vedat Coskun, Kerem Ok, and Busra Ozdenizci. *Near field communication (NFC): From theory to practice*. John Wiley & Sons, 2011.

[13] Amitava Dutta-Roy. Networks for homes. *IEEE spectrum*, 36(12):26–33, 1999.

[14] Miro Enev, Sidhant Gupta, Tadayoshi Kohno, and Shwetak N. Patel. Televisions, video privacy, and powerline electromagnetic interference. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 537–550, New York, NY, USA, 2011. ACM.

[15] Hendrik C Ferreira, Lutz Lampe, John Newbury, and Theo G Swart. *Power line communications: theory and applications for narrowband and broadband communications over power lines*. John Wiley & Sons, 2011.

[16] Klaus Finkenzeller. *RFID handbook: fundamentals and applications in contactless smart cards, radio frequency identification and near-field communication*. John wiley & sons, 2010.

[17] Henry C Forbes. Re-radiation from tuned antenna systems. *Proceedings of the Institute of Radio Engineers*, 13(3):363–382, 1925.

[18] Juan Carlos Rodriguez Guerra. Electric field energy harvesting from medium voltage power lines. *School of Engineering, College of Science, Engineering and Health, RMIT University Australia*, 208, 2017.

[19] Manoj Gulati, Farshid Salemi Parizi, Eric Whitmire, Sidhant Gupta, Shobha Sundar Ram, Amarjeet Singh, and Shwetak N. Patel. Capharvester: A stick-on capacitive energy harvester using stray electric field from ac power lines. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 2(3), September 2018.

[20] Sidhant Gupta, Matthew S. Reynolds, and Shwetak N. Patel. Electrisense: Single-point sensing using emi for electrical event detection and classification in the home. In *Proceedings of the 12th ACM International Conference on Ubiquitous Computing*, UbiComp '10, page 139–148, New York, NY, USA, 2010. Association for Computing Machinery.

[21] Vikram Gupta, Arvind Kandhalu, and Ragunathan (Raj) Rajkumar. Energy harvesting from electromagnetic energy radiating from ac power lines. In *Proceedings of the 6th Workshop on Hot Topics in Embedded Networked Sensors*, HotEmNets '10, New York, NY, USA, 2010. Association for Computing Machinery.

[22] David Halliday, Robert Resnick, and Jearl Walker. *Fundamentals of physics*. John Wiley & Sons, 2013.

[23] Mehrdad Hessar, Ali Najafi, and Shyamnath Gollakota. Netscatter: Enabling large-scale backscatter networks. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 271–284, 2019.

[24] Bryce Kellogg, Aaron Parks, Shyamnath Gollakota, Joshua R Smith, and David Wetherall. Wifi backscatter: Internet connectivity for rf-powered devices. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, pages 607–618, 2014.

[25] Yu-Ju Lin, Haniph A Latchman, Minkyu Lee, and Srinivas Katar. A power line communication network infrastructure for the smart home. *IEEE wireless communications*, 9(6):104–111, 2002.

[26] DJR Martin. Leaky-feeder radio communication: A historical review. In *34th IEEE Vehicular Technology Conference*, volume 34, pages 25–30. IEEE, 1984.

[27] Marcel Nassar, Jing Lin, Yousof Mortazavi, Anand Dabak, Il Han Kim, and Brian L Evans. Local utility power line communications in the 3–500 khz band: Channel impairments, noise, and standards. *IEEE signal processing magazine*, 29(5):116–127, 2012.

[28] Amir Mehdi Pasdar, Ismail H Cavdar, and Yilmaz Sozer. Power-line impedance estimation at fcc band based on intelligent home appliances status detection algorithm through their individual energy and impedance signatures. *IEEE Transactions on power delivery*, 29(3):1407–1416, 2014.

[29] Shwetak N. Patel, Thomas Robertson, Julie A. Kientz, Matthew S. Reynolds, and Gregory D. Abowd. At the flick of a switch: Detecting and classifying unique electrical events on the residential power line. In *Proceedings of the 9th International Conference on Ubiquitous Computing*, UbiComp '07, page 271–288, Berlin, Heidelberg, 2007. Springer-Verlag.

[30] Shwetak N. Patel, Erich P. Stuntebeck, and Thomas Robertson. Pl-tags: Detecting batteryless tags through the power lines in a building. In *Proceedings of the 7th International Conference on Pervasive Computing*, Pervasive '09, page 256–273, Berlin, Heidelberg, 2009. Springer-Verlag.

[31] Shwetak N. Patel, Khai N. Truong, and Gregory D. Abowd. Powerline positioning: A practical sub-room-level indoor location system for domestic use. In Paul Dourish and Adrian Friday, editors, *UbiComp 2006: Ubiquitous Computing*, pages 441–458, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[32] Shwetak N Patel, Khai N Truong, and Gregory D Abowd. Powerline positioning: A practical sub-room-level indoor location system for domestic use. In *International Conference on Ubiquitous Computing*, pages 441–458. Springer, 2006.

[33] Nicodimus Retdian and Takeshi Shima. N-path notch filter with a 43-db notch depth improvement for power line noise suppression. In *2016 International symposium on electronics and smart devices (ISESD)*, pages 184–187. IEEE, 2016.

[34] Anthony Rowe, Vikram Gupta, and Ragunathan Rajkumar. Low-power clock synchronization using electromagnetic energy radiating from ac power lines. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, pages 211–224, 2009.

[35] Anthony Rowe, Vikram Gupta, and Ragunathan (Raj) Rajkumar. Low-power clock synchronization using electromagnetic energy radiating from ac power lines. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys '09, page 211–224, New York, NY, USA, 2009. Association for Computing Machinery.

[36] Erich P Stuntebeck, Shwetak N Patel, Thomas Robertson, Matthew S Reynolds, and Gregory D Abowd. Wideband powerline positioning for indoor localization. In *Proceedings of the 10th international conference on Ubiquitous computing*, pages 94–103, 2008.

[37] Erich P Stuntebeck, Thomas Robertson, Gregory D Abowd, and Shwetak N Patel. Using in-home power lines to extend the range of low-power wireless devices. Technical report, Georgia Institute of Technology, 2009.

[38] Nima Taherinejad, Roberto Rosales, Shahriar Mirabbasi, and Lutz Lampe. On the design of impedance matching circuits for vehicular power line communication systems. In *2012 IEEE International Symposium on Power Line Communications and Its Applications*, pages 322–327. IEEE, 2012.

[39] Vamsi Talla, Mehrdad Hessar, Bryce Kellogg, Ali Najafi, Joshua R Smith, and Shyamnath Gollakota. Lora backscatter: Enabling the vision of ubiquitous connectivity. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 1(3):1–24, 2017.

[40] PA Janse Van Rensburg and Hendrik C Ferreira. Design of a bidirectional impedance-adapting transformer coupling circuit for low-voltage powerline communications. *IEEE Transactions on Power Delivery*, 20(1):64–70, 2005.

[41] PA Janse van Rensburg, Umer Izhar, and DMG Preethichandra. A novel plc impedance conditioning technique for quasi-common mode power-line

antenna injection. In *2018 IEEE 27th International Symposium on Industrial Electronics (ISIE)*, pages 677–682. IEEE, 2018.

[42] Jingxian Wang, Junbo Zhang, Rajarshi Saha, Hao-jian Jin, and Swarun Kumar. Pushing the range limits of commercial passive rfids. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 301–316, 2019.

[43] Ron Weinstein. Rfid: a technical overview and its application to the enterprise. *IT professional*, 7(3):27–33, 2005.

[44] Vivek Yenamandra and Kannan Srinivasan. Vidyut: exploiting power line infrastructure for enterprise wireless networks. *ACM SIGCOMM Computer Communication Review*, 44(4):595–606, 2014.

[45] Larry Yonge, J. Abad, K. Afkhamie, L. Guerrieri, S. Katar, Hidayat Lioe, P. Pagani, R. Riva, D. Schneider, and A. Schwager. An overview of the homeplug av2 technology. *J. Electr. Comput. Eng.*, 2013:892628:1–892628:20, 2013.

[46] Larry Yonge, Jose Abad, Kaywan Afkhamie, Lorenzo Guerrieri, Srinivas Katar, Hidayat Lioe, Pascal Pagani, Raffaele Riva, Daniel M Schneider, and Andreas Schwager. An overview of the home-plug av2 technology. *Journal of Electrical and Computer Engineering*, 2013, 2013.

[47] Pengyu Zhang, Dinesh Bharadia, Kiran Joshi, and Sachin Katti. Hitchhike: Practical backscatter using commodity wifi. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*, pages 259–271, 2016.

[48] Tian Zhou, Yue Zhang, Xinlei Chen, Khaild M Mosalam, Hae Young Noh, Pei Zhang, and Lin Zhang. P-loc: a device-free indoor localization system utilizing building power-line network. In *Adjunct Proceedings of the 2019 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2019 ACM International Symposium on Wearable Computers*, pages 611–615, 2019.

Passive DSSS: Empowering the Downlink Communication
for Backscatter Systems

Songfan Li[1], Hui Zheng[1], Chong Zhang[1], Yihang Song[1], Shen Yang[1], Minghua Chen[1], Li Lu[1] and Mo Li[2]
[1]*University of Electronic Science and Technology of China (UESTC),*
[2]*Nanyang Technological University (NTU)*

Abstract

The uplink and downlink transmissions in most backscatter communication systems are highly asymmetric. The downlink transmission often suffers from its short range and vulnerability to interference, which limits the practical application and deployment of backscatter communication systems. In this paper, we propose *passive DSSS* to improve the downlink communication for practical backscatter systems. Passive DSSS is able to increase the downlink signal-to-interference-plus-noise ratio (SINR) by using direct sequence spread-spectrum (DSSS) techniques to suppress interference and noise. The key challenge lies in the demodulation of DSSS signals, where the conventional solutions require power-hungry computations to synchronize a locally generated spreading code with the received DSSS signal, which is infeasible on energy-constrained backscatter devices. Passive DSSS addresses such a challenge by shifting the generation and synchronization of the spreading code from the receiver to the gateway side, and therefore achieves ultra-low power DSSS demodulation. We prototype passive DSSS for proof of concept. The experimental results show that passive DSSS improves the downlink SINR by 16.5 dB, which translates to a longer effective downlink range for backscatter communication systems.

1   Introduction

Passive communication is expected to be a promising connectivity paradigm for building Internet of things (IoT) due to its ultra-low power and low-cost features. Significant efforts have been put into improving the backscatter link of passive communication, in terms of range [43, 49, 53–56], robustness [23, 33, 59], and interoperability with commercial radios like Wi-Fi [25, 26] and LoRa [43, 49]. In particular, unlike RFID that communicates while harvesting RF power, advanced backscatter communication decouples the communication from RF power harvesting [50], and thus becomes a low-power communication solution for IoT devices with longer communication range and limited power supplies (*e.g.*, with small batteries or energy harvesting modules).
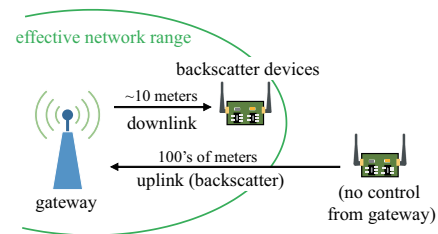

Figure 1: Passive DSSS addresses the asymmetric link issue by empowering the downlink with interference resilience.

Practical deployment of backscatter communication systems faces major challenges in communication asymmetry. As Fig. 1 shows, the backscatter link (uplink) of passive communication and its downlink are often highly asymmetric, where the downlink transmission has poorer performance than the uplink in terms of its range and interference resilience. In most backscatter communication systems, we have a powerful gateway that can be comprehensively designed to demodulate and decode noisy signals on the backscatter uplink. On the downlink, however, receivers on backscatter devices are low power operated and vulnerable to interference and noise.

In practice, IoT devices often need downlink controls. For example, when a communication collision occurs among multiple backscatter devices, the gateway needs to send downlink controls to mediate those devices so as to resolve the collision. Other useful downlink transmissions include scheduling transmissions [15], synchronizing networks [12], sending wake-up packets [22], controlling sensors [28] and implementing over-the-air (OTA) firmware update [61] and so on. The poor communication performance of the downlink becomes a major limit to the range of the backscatter communication system.

In this paper, we ask whether it is possible to significantly improve the downlink resilience to interference. Essentially, for improving the downlink communication, we need to increase the signal quality in terms of signal-to-interference-plus-noise ratio (SINR). Rather than increasing the trans-

mission power of the downlink signal which is essentially limited by radio spectrum regulators like FCC [7] and may lead to undesired extra interference, in this paper we look for a solution that suppresses the interference on the other hand. We make use of spread-spectrum modulation schemes (*e.g.*, DSSS) to suppress interference and noise. However, to the best of our knowledge, none of the existing spread-spectrum schemes can directly work for the downlink of backscatter systems. Specifically, those spread-spectrum systems require the receiver to incorporate a high frequency oscillator to correlate with the received spread-spectrum signal [6], which inevitably incurs high power consumption that is undesirable for energy-constrained backscatter devices.

We present passive DSSS, the first direct sequence spread-spectrum (DSSS) technique for passive communication to suppress interference and noise. As Fig. 2(a) shows, in conventional DSSS communication, the receiver requires complex receiver circuitry and expensive computations for synchronization between the received DSSS signal and locally generated spreading code. Specifically, the DSSS transmitter spreads the frequency spectrum of baseband signals across a wider band by modulation with a pseudorandom spreading code. The DSSS receiver strips off the spreading code and retrieves the original baseband by de-spreading (demodulating) the received signal with a synchronized replica of the spreading code. The de-spreading process requires computationally expensive synchronization between the DSSS signal and local spreading code, which is infeasible on backscatter devices.

To address the challenges associated with the complicated de-spreading of DSSS demodulation, the proposed solution offloads the spreading code generation and synchronization from the backscatter device to the gateway side. As shown in Fig. 2(b), the gateway transmits the DSSS signal and the spreading code reference simultaneously via two separate channels to the receiver. As the spreading code is inherently synchronized with the DSSS signal at the transmitter side, there is no more need for synchronization at the receiver side for de-spreading. The spreading code can be stripped off by combining the two channels together after removing the carrier waves. To achieve passive DSSS in practice, we need to address the following three technical challenges.

Challenge-1. Conventional DSSS systems suppose that the receiver can estimate phase information of the channel. However, obtaining phase information needs the use of a local oscillator operating at the carrier frequency, which is infeasible on backscatter devices. In passive DSSS, we leverage the envelope of an RF carrier to convey the DSSS signal (§ 3). Specifically, the gateway transmits the spreading code by modulating the amplitude of the carrier, while the baseband signal is communicated by modulating the phase difference between the synchronized spreading codes in each individual channel. The receiver reconstructs the baseband by comparing the two spreading codes.
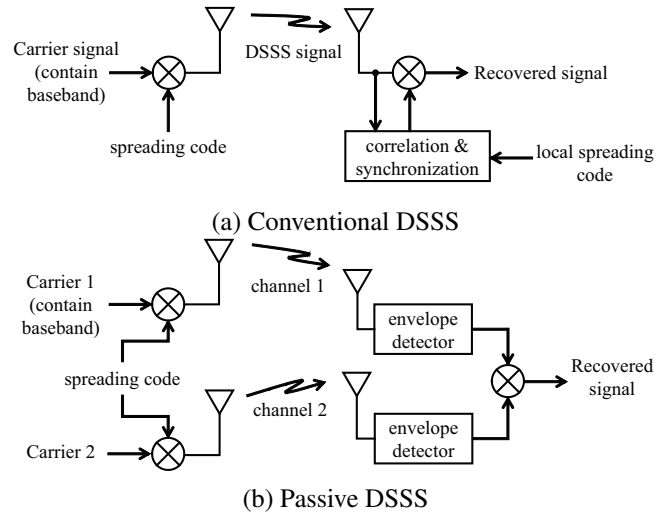

(a) Conventional DSSS

(b) Passive DSSS

Figure 2: Passive DSSS shifts the synchronization of the spreading code to a gateway, thereby enabling backscatter devices to demodulate DSSS transmission.

Challenge-2. The two separate channels to convey the DSSS signal and the spreading code may experience different interference effects in practice. In conventional DSSS receivers, interference is suppressed by multiplying with a spreading code in the de-spreading process. In passive DSSS, however, the de-spreading process performs the multiplication of two channel signals, thus leading to an interference composition (the product of interference signals from two channels). We explore the fact that the interference signals between the separate channels are often independent of each other and propose a solution (§ 4) to suppress the interference composition by calculating the cross-correlation between the two interfered signals.

Challenge-3. Conventional correlation operation requires power-consuming digitization and signal processing which cannot be accommodated on backscatter devices. To overcome this challenge, we design an interference cancellation circuit with analog components to compute the cross-correlation (§ 5). As the received signals from two channels are already multiplied with each other in the de-spreading process, the interference cancellation circuit mainly performs the integration of the signal output from the de-spreading process in analog domain. The original baseband signal predominates in the output of the cancellation circuit.

We build a prototype system for proof of concept. We implement the gateway with NI USRP 2922 and implement the passive DSSS receiver with commercial off-the-shelf (COTS) hardware components. The passive DSSS receiver consumes 166.5 μW power when demodulating the DSSS signals robustly with 1 MHz bandwidth. We evaluate the prototype system with realistic communication environments with RFID and LoRa interference. We also conduct stress-testing experiments to examine the performance of passive DSSS in terms

of BER with different noise and interference levels. The results show that passive DSSS can gain an SINR improvement of 16.5 dB over conventional receivers on backscatter devices. The higher SINR translates to a longer effective downlink range for backscatter systems. Our experimental results show that the effective downlink range is extended by up to 52 m with 20 dBm transmission power, which is 3x longer than what can be offered by conventional receivers.

2 Preliminary

This section describes the preliminary knowledge before we come to the design of passive DSSS, including descriptions of the reason why DSSS techniques can suppress interference, and the synchronization problem to achieve DSSS on backscatter devices.

2.1 A Primer for Conventional DSSS

DSSS is a spread-spectrum modulation technique primarily used for reducing overall signal interference. While DSSS is also employed to achieve concurrent transmissions by the code-division multiple access (CDMA) method in wireless systems (*e.g.*, cellular and GPS), this paper aims to exploit the interference resilience of DSSS to improve the downlink of backscatter communication systems.

Non-spread spectrum wireless communication transmits baseband information by modulating an RF carrier, which can be treated as a narrowband signal that is easily disturbed by any other interferers in the same band. The idea behind spread-spectrum is to use a wider bandwidth than the original baseband (typically 10–60 dB), and therefore diffuses the information across a larger bandwidth, which allows recovery of the transmitted signal even when a part of the spectrum is significantly impaired by narrowband interference.

To achieve DSSS transmission, spread-spectrum modulation is applied on top of conventional modulation by multiplication with the corresponding spreading code before transmission. The spreading code is a pseudorandom sequence with a much higher data rate than the baseband. The produced signal stream thus has a higher data rate and occupies a wider signal bandwidth.

A despreading operation at the receiver side reconstitutes the information in its original bandwidth. The received signal is multiplied by a replica of the spreading code $\hat{c}(t)$ in order to regenerate the original data. The despreading operation can be mathematically represented as:

$$
\begin{aligned}
& \overbrace{[b(t)c(t)+I(t)]}^{\text{received signal}} \cdot \hat{c}(t) \\
= {} & \underbrace{b(t)c(t)\hat{c}(t)}_{\text{despreading}} + \underbrace{I(t)\hat{c}(t)}_{\text{suppression}}
\end{aligned}
\tag{1}
$$

where $b(t)$ is the original baseband signal, $c(t)$ represents the spreading code, and $I(t)$ refers to the interference. If $\hat{c}(t)$ is precisely synchronized with $c(t)$, we say $\hat{c}(t) = c(t)$. Since $c(t) = \pm 1$, the product $c(t)\hat{c}(t)$ is unity when they are synchronized, so that the first term (despreading) is equal to the desired baseband $b(t)$. In the second term (suppression), the interference signal is multiplied by the spreading code, which spreads the interference spectrum. As the spread interference features much higher data rates than the baseband signal, we can remove the interference with a low pass filter (LPF).

2.2 Problem of DSSS Synchronization

If the spreading code is synchronized, the output of a despreading operation will be a correct baseband signal. Otherwise, the received spread-spectrum signal cannot be demodulated correctly because $c(t)\hat{c}(t)$ will be a noise-like rapidly moving code which hides the baseband signal. It is difficult for a receiver to accurately recover the slow baseband signal without having an exact replica of the spreading code.

The synchronization process, however, requires that the DSSS receiver performs phase estimation and intensive computation, neither of which is suitable for backscatter devices. Specifically, the synchronization is often accomplished with two steps, *i.e.*, *acquisition* and *tracking*. First, *acquisition* determines the phase of the spreading sequence in the received signal. The *tracking* step then continuously maintains the best alignment between the locally generated spreading code and the received DSSS signal. Although prior works have proposed low-power DSSS techniques [6, 13, 21, 34, 36] to reduce the power consumption for synchronization, they are designed for active radios in which power-starving oscillators and analog-to-digital converters (ADC) are employed in their architectures. Those components typically consume at the scale of *mW*, which is still higher than what can be afforded on backscatter devices.

There are also prior works that apply DSSS in RFID systems. Arthaber *et al.* [2] adopt DSSS to increase the tag-to-reader communication range for RFID, in which each backscatter tag is assigned a unique spreading code to encode the backscatter data. Some works employ CDMA to address the problem of collisions [38, 59, 62] in RFID networks. The code is designed to be orthogonal and thus allows an RFID reader to decode the transmitted data even in presence of collisions. Those works, however, are designed for the uplink transmission from the backscatter tag to the reader, where the power consuming and computationally expensive synchronization process is performed at the reader side.

A recent work μcode [41] proposes a CDMA-like method to turbocharge tag-to-tag backscatter communication. Instead of using a pseudorandom spreading code, μcode leverages a periodic signal with an alternating one-zero sequence in the transmission to avoid the synchronization at the receiver side. However, the periodic signal is unable to spread the baseband
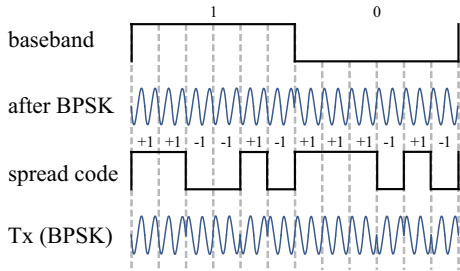
(a) Conventional DSSS modulation
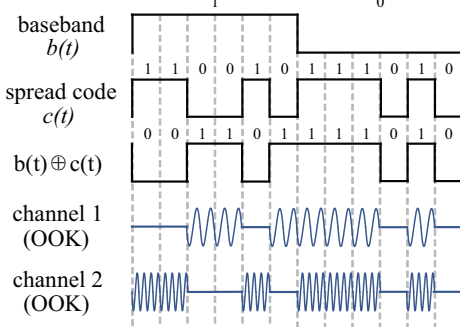
(b) Passive DSSS modulation

Figure 3: Comparison between conventional DSSS and passive DSSS transmission

signal over a wide frequency band and thus cannot offer the desired anti-interference feature. In conclusion, in order to suppress interference, passive DSSS has to make use of the spreading code in the downlink transmission and address the synchronization problem at the receiver on backscatter devices.

3   Passive DSSS Modulation and Transmission

We design a passive DSSS modulation scheme for spread-spectrum signal transmissions without phase estimation on the receiver side. The basic intuition comes from an observation, where receivers on backscatter devices can effectively detect the amplitude information from the incoming signal with simple envelope detection. The envelope detector can be built with passive components such as resistors, capacitors and diodes, and is thus ultra-low power in nature. The designed passive DSSS may utilize the envelope of the carrier signal to convey synchronized spreading codes in two individual channels and leverage the phase difference between the spreading codes to represent the baseband information. Specifically, the gateway conveys an unmodulated spreading code $c(t)$ via one channel (channel 2 as illustrated in Fig. 3(b)) by modulating the envelope of the RF carrier with on-off keying (OOK) modulation. In the other channel (channel 1), the gateway simultaneously conveys a modulated spreading code $b(t) \oplus c(t)$ where the baseband modulates the phase of the spreading code. With the above, the passive DSSS receiver can reconsti-

Table 1: XOR operation works as BPSK modulation to the spreading code $c(t)$.

	$b(t)$	$c(t)$	$b(t) \oplus c(t)$
no phase change $\{$	0	0	0
	0	1	1
π phase change $\{$	1	0	1
	1	1	0

tute the baseband by comparing the phase shift between the received signals from the two channels.

Figure 3 presents a comparison between the modulation schemes of conventional and passive DSSS. To convey the baseband information by the phase difference, passive DSSS employs an XOR operation to apply the binary phase shift keying (BPSK) modulation to the spreading code. Table 1 shows the truth table of two inputs and their XOR output. We see that when $b(t) = 0$, the XOR output has no change to $c(t)$, whereas when $b(t) = 1$ the output is inverted, performing a π phase change to the spreading signal $c(t)$. Therefore, the two spreading codes carried in the two channels are $b(t) \oplus c(t)$ (channel 1) and $c(t)$ (channel 2) respectively.

At the receiver side, the passive DSSS transmission is demodulated with two steps. First, the spreading code carried in each channel is obtained with envelope detection. Second, the baseband signal can be recovered by despreading, where an XOR gate is employed to perform symmetrical BPSK demodulation by combining the spreading codes from both channels. Since the spreading codes $b(t) \oplus c(t)$ and $c(t)$ are synchronized by transmission, the XOR gate derives $b(t) \oplus c(t) \oplus c(t)$ and retrieves the baseband $b(t)$.

4   Interference Suppression

We have described the basic idea of passive DSSS. In this section, we discuss the rationale of interference suppression provided by passive DSSS.

4.1   Realistic Interference Signals

ISM bands are often approved for license-free which can be used without a government license. This means that multiple types of radio applications may share the same radio spectrum and interfere with each other. This issue increasingly challenges the receivers on backscatter devices because their poor performance cannot provide interference-tolerant downlink communication.

We survey ambient RF signals in 915MHz and 2.4GHz ISM bands in practice at an office and a shopping mall, respectively. In experiments, we use the Keysight N9912A RF handheld analyzer to collect the in-band signals and observe the spectrum of those signals. To quantitatively identify which signals are potential interference, we run two wireless sys-

(a) Experiment setup   (b) Backscatter harmonics   (c) Adjacent in-band signals
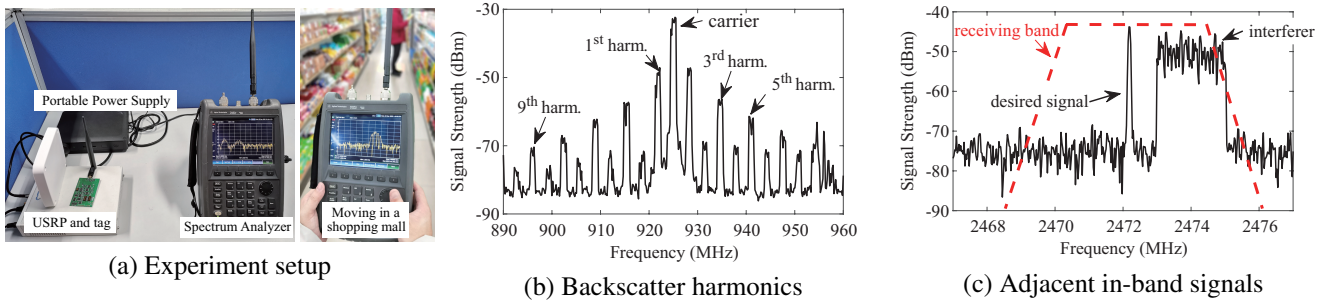
Figure 4: (a) RF interference measurement setup. Typical interference signals (b) and (c) in ISM bands.

tems in our experiments, where one is a 915MHz backscatter system with 1 Mbps chirp modulation, and the other is for 200 kbps downlink ASK transmission to a 2.4GHz backscatter tag. Figure 4(a) presents the experiment setup. With an observation over one week, we summarize three common interference types observed in below.

- *Harmonics:* Backscatter signals are generated by switching the antenna impedance between different loads. The harmonics when backscattering may result in interference in adjacent frequencies, which even spreads over a wide band (Fig. 4(b)). Although prior works exploited solutions including single sideband backscatter [18] and backscatter harmonic cancellation [49] to reduce the harmonic interference, some other backscatter systems may intentionally create stronger harmonics to convey information such as cross frequency communication [1] and localization [35]. Therefore, the harmonics may have a strong presence in ISM bands, leading to disturbance to backscatter networks.

- *Adjacent in-band signals:* The second type of interference is the disturbance from adjacent band signals. Fig. 4(c) presents an example where the backscatter tag receives the adjacent signal within the receiving band, which causes significant interference to the downlink transmission. Although some digital band-pass filtering techniques may be applied to remove the adjacent interference, they are often computationally expensive and thus unfits for backscatter devices.

- *Overlapping signals:* The third case is when two signals overlap in frequency and become interference to each other. Although existing RFID systems provide the frequency hopping mode, it is only available for preventing interference on the uplink in dense reader environments, and cannot resist interference on the downlink.

As a consequence, existing receivers on backscatter devices do not have an effective solution against the realistic interference existing in ISM bands. In the next portion, we illustrate how passive DSSS suppresses the interference in principle.

4.2 Interference Suppression in Passive DSSS

We take all of the above interference signals into account. Mathematically, the interference signal is added to the received signal. We denote the received signal with interference at channel 1 by $S_1(t) + I_1(t)$, where $S_1(t)$ represents the DSSS signal envelope transmitted from the gateway, and $I_1(t)$ is a bipolar interference effect on the signal envelop. If $I_1(t)$ is a negative value, a bit "$1 \rightarrow 0$" error may occur, and vice versa. Similarly, the received signal at channel 2 can be represented as $S_2(t) + I_2(t)$.

As illustrated in § 3, the received envelopes from the two separate channels are input into an XOR gate for de-spreading and interference suppression. Such a process can be formulated as:

$$[S_1(t) + I_1(t)] \oplus [S_2(t) + I_2(t)]$$

As XOR operation for binary signals can be derived by $a \oplus b = a + b - 2ab$, the above process can be written as:

$$\underbrace{S_1(t) + S_2(t) - 2S_1(t) \cdot S_2(t)}_{\text{① despreading}}$$
$$+ \underbrace{[1 - 2S_2(t)] \cdot I_1(t) + [1 - 2S_1(t)] \cdot I_2(t)}_{\text{② suppression}}$$
$$- \underbrace{2I_1(t) \cdot I_2(t)}_{\text{③ product of interference}} \quad (2)$$

where the first term ① is equal to $S_1(t) \oplus S_2(t)$, thereby performing the de-spreading operation to regenerate the baseband signal.

The second term ② contains the interference signals $I_1(t)$ and $I_2(t)$ from the two channels, which however are suppressed by the spreading code from the other channel, respectively. To understand this, we say that $S_1(t)$ and $S_2(t)$ are polar envelope signals (between 0 and 1) that comprise the spreading code, while $I_1(t)$ and $I_2(t)$ are bipolar signals (between -1 and +1) representing the possible information disturbance (bit $1 \rightarrow 0$ or bit $0 \rightarrow 1$). Therefore, the term $1 - 2S_1(t)$ is equivalent to converting $S_1(t)$ to a bipolar signal and inverting its polarity, which does not change the spreading code in terms of data rate. Hence, the interference effects in

(a) Interference signals
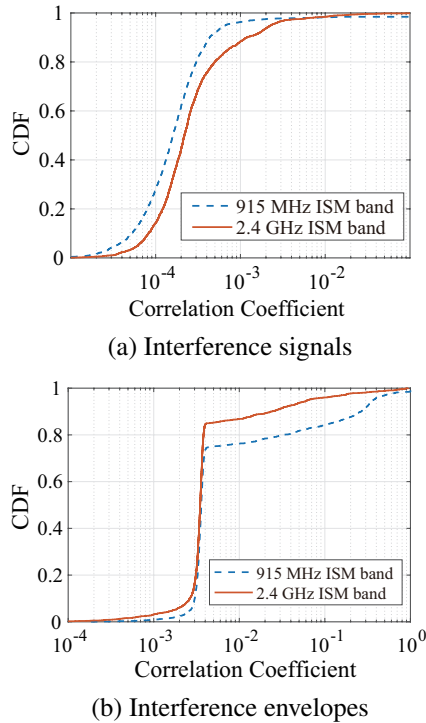

(b) Interference envelopes

Figure 5: (a) The correlation of the interference signals from two individual channels. (b) The correlation of their envelopes.

the second term are suppressed by the multiplication with the bipolar spreading code.

However, the third term ③ does not contain the spreading code. The product of the two interference compositions is still an interference signal to baseband. To mitigate the interference, existing solutions can be categorized into the following three types. First, the receiver may divide the received signal bandwidth into multiple sub-bands, and then position sub-band notch filters (band-stop) to suppress the interferer [8, 9, 42]. Second, statistical methods may be used at the receiver to average the interference signal over multiple symbols [63]. Third, feedback loop mechanisms may be used to enhance the desired signal by iterative correlation [10], in which each loop iteration makes the desired signal cleaner. All of the above methods, however, require power-starving digital signal processing [37] which is not suitable for backscatter devices.

Fortunately, we observe that the two interference signals in each individual channel are almost independent of each other. To verify this, we conduct a one-week measurement in real environments. In 915 MHz ISM band, we turn on the backscatter tag in Fig. 4(b) to emulate interference from backscatter networks. In 2.4 GHz ISM band, we choose an in-door environment where two Wi-Fi routers and a number of Bluetooth devices operate. At least ten Wi-Fi channels and sixteen Bluetooth channels have strong signal presence.
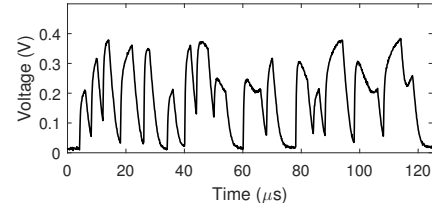

Figure 6: The signal envelope distorted by interference is full analog.

We employ a USRP to arbitrarily pick two separate channels with 1 MHz bandwidth in the ISM bands respectively, and compute the correlation between the interference signals over the two channels. Figure 5 plots the results of the correlation between the two interference signals and their envelopes. If an interference signal resembles the other one, the correlation coefficient will be close to 1. We see that the overall correlation remains statistically very low, indicating almost uncorrelated interference signals.

Given the low interference correlation, we can cancel the interference signals after the de-spreading by computing the correlation of the two interference compositions in the third term, given by:

$$\int_0^T I_1(t) \cdot I_2(t) \, dt \approx 0 \qquad (3)$$

where T is the duration of the de-spreading process that provides the calculation of the dot product operation. In section § 6.4.2, we evaluate the performance of passive DSSS over the interference of different correlation coefficients.

5   Low Power DSSS De-spreading

In this section, we present the hardware design to implement the design rationale described in 4.2. At a high level, the passive DSSS receiver first obtains the envelopes of the spread-spectrum signals and inputs the envelope waveforms into an XOR gate for despreading. The receiver further employs a hybrid analog-digital computing circuit to derive the signal correlation in order to suppress the interference.

5.1   Despreading Process

We combine the two envelope signals with an XOR gate to perform the despreading process. However, when the transmission is disturbed by interference in the wireless channels, the signal envelope captured from the air is fully analog as shown in Fig. 6. The XOR gate is a digital component and thus only accepts digital signal inputs, meaning that the passive DSSS receiver has to digitize the envelope signals before the despreading process. Conventional DSSS receivers in active radios realize digitization with two steps — sampling the
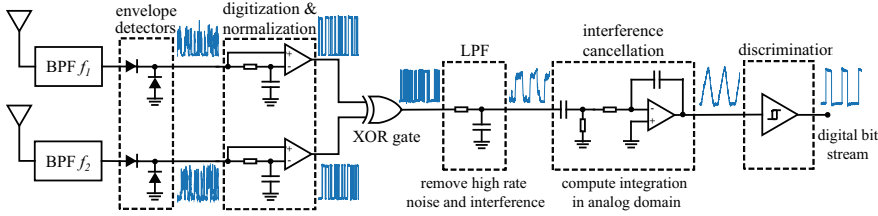
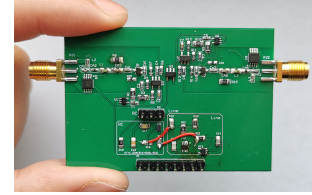Figure 7: Passive DSSS receiver hardware design.

Figure 8: Passive DSSS receiver prototype.

voltage signal with a precise ADC (*e.g.*, 8-12 bit) and then discriminating the binary value according to a decision-making criterion (*e.g.*, maximum likelihood estimation). The two steps involved in active radios are not feasible on backscatter devices due to the high power consumption. Additionally, due to frequency selective fading on the two separate channels, the received envelope signals may experience different amplitude attenuation, meaning that the two-channel signals should not be directly combined without normalization.

Passive DSSS receivers adopt an ultra-low power digitization method, where a one-bit ADC (*i.e.*, comparator) is used instead of a high precision ADC to threshold the analog envelope signals, and discriminate the digital value by its moving average. Specifically, the comparator compares the analog envelope voltage and the moving average of the envelope in order to distinguish between the two binary levels at the output of the envelope detector. The moving average is automatically computed by an RC low-pass circuit at one input of the comparator, and thus creates a dynamic threshold for digitizing different input signal amplitudes. In addition, the one-bit ADC also normalizes the received signal from each channel, so the output of digitization can be directly supplied to the XOR gate for DSSS despreading.

Figure 7 gives the hardware design of the passive DSSS receiver. The two receiver branches are designed symmetrically in order to maintain the synchronization of the received spread-spectrum signals between the two channels. The bandpass filters (BPF) are employed to achieve frequency band selection that isolates the signal from the other channel. The envelope detector then removes the carrier waves of the incoming signals in both channels and outputs the envelope waveforms. Further, the one-bit digitization circuit digitizes the disturbed envelopes and sends them into the XOR gate for the despreading process.

5.2 Analog-Digital Correlation Computation

Another problem to achieve the passive DSSS receiver is that computing correlation involves intensive computations [14] that are unacceptable to backscatter devices. To address this, we design a hybrid analog-digital computing circuit (see Fig. 7) to compute the correlation for interference suppression with ultra-low power. At a high level, the XOR gate performs a digital operation which comprises the multiplication of the

two interference signals (Eq. 2, the third term). Therefore, we further design an analog interference cancellation circuit to compute the integration of the XOR output to suppress the interference composition in analog domain.

To understand our design, we recall the despreading process described in § 4.2. The despreading result at the output of the XOR gate includes three basic compositions: the desired baseband, the suppressed interference and the product of the two interference signals. We first remove the suppressed interference using an RC-based LPF because they contain the spreading codes that have much higher data rates (frequencies) than the baseband. Then, the interference cancellation circuit performs an integration operation over the rest of the signals in order to remove the interference product composition. The interference cancellation process can be represented as:

$$
\begin{aligned}
&\int_0^T \left[S_1(t) \oplus S_2(t)\right] - 2\left[I_1(t) \cdot I_2(t)\right] \mathrm{d}t \\
={}& \int_0^T b(t)\,\mathrm{d}t - 2\int_0^T I_1(t) \cdot I_2(t)\,\mathrm{d}t \\
\approx{}& \int_0^T b(t)\,\mathrm{d}t
\end{aligned}
$$

According to Eq. 3, the product integration of the interference signals is negligible due to their low correlation. Therefore, the result of the interference cancellation is approximately equal to the integration of the baseband signal. Further, as the baseband comprises ones and zeros that are represented by logical high and low voltages, we can discriminate the baseband data according to the energy difference between ones and zeros. Finally, the discrimination output is a digital bit stream representing the baseband data and can be directly handled by the MCU of the device.

The interference cancellation circuit incorporates a DC block capacitor and an analog linear integrator. The DC block capacitor precedes the integrator in order to convert the polarity of the output signals of LPF from polar to bipolar. The bipolar signals facilitate the subsequent integrator in performing integration.
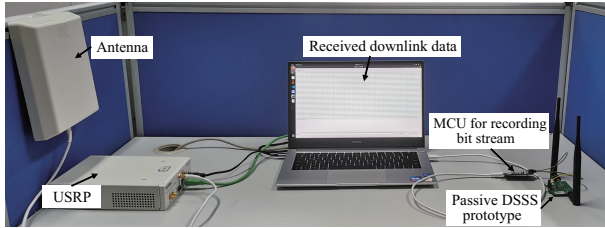
Figure 9: Experimental setup.

Figure 10: Performance comparison. We choose the case of conventional (2 channels) as the baseline.

6 Evaluation

We describe the implementation experience of the passive DSSS prototype in § 6.1 and discuss its power consumption. In § 6.2, we evaluate the communication performance of passive DSSS and decide the baseline for subsequent evaluations. Further, we evaluate passive DSSS with the interference from real world environments and also estimate the communication range of passive DSSS in § 6.3. We evaluate the performance boundary of passive DSSS with stress-test in § 6.4. Finally, we discuss the application case study in § 6.5.

6.1 Implementation

We implement our passive DSSS receiver prototype (Fig. 8) with commercial off-the-shelf (COTS) components. We employ the SAW BPF filters from NMRF (one is 890 MHz∼915 MHz, the other is 920 MHz∼925 MHz) to distinguish between the two channels. The envelope detectors adopt the Schottky diodes HSMS-285C. As the filters and diodes are passive components, they do not consume power. Further, the 1-bit digitization circuit is built with the NCS2200 comparator. The pair of digitization circuits consume 20 μA current. Next, the XOR gate is implemented using SN74LV1T86 from Texas Instruments (1.25 μA current consumption), and the cutoff frequency of the RC-based LPF is set to twice the baseband frequency. Moreover, the interference cancellation circuit employs a 10 nF DC block capacitor and comprises a power-efficient TSV6390 operational amplifier in the integrator circuit (50 μA current consumption). Finally, we use the nano-power comparator MAX40000 (12 μA) made by Maxim Integrated for the final discrimination. The output of the discriminator is a digital bit stream that can be directly channeled to an MCU. Since the MCU does not participate in the demodulation process of passive DSSS, we do not consider MCU's power consumption. According to our measurement study, the passive DSSS receiver totally consumes 166.5 μW power when operating with 1 MHz signal bandwidth and 2 V supply voltage. When commercially adopted, the power consumption can be further reduced by application-specific integrated circuit (ASIC) implementation [26, 39, 60].
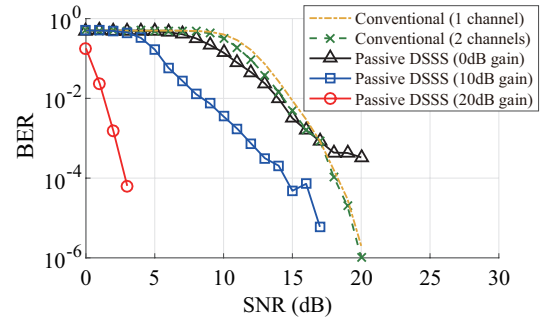
6.2 Performance and Baseline

Figure 9 depicts the experiment setting where we employ a USRP to send downlink transmissions to our prototype. The received data stream is forwarded by the onboard MCU to the laptop for bit error rate (BER) analysis. While passive DSSS theoretically supports arbitrary digital coding schemes of the baseband, we adopt pulse-interval encoding (PIE) in the experiments for its wide adoption in many passive radios like RFID. To evaluate BER, the USRP transmits one million test bits on the downlink during each time of the measurement to derive the average.

We evaluate the communication performance in terms of BER with noise. Passive DSSS makes use of two channels, whereas conventional receivers on backscatter devices only use one channel for the downlink transmission. In our experiments, we consider both conventional receivers and passive DSSS receivers. We note that the passive DSSS receiver has two symmetric antenna branches, in which each branch is a conventional receiver. Thus, we can employ each antenna branch of the passive DSSS receiver to act as a conventional receiver that shares the same hardware and layout with passive DSSS. The difference is that the conventional receivers receive non-DSSS signals from the gateway. Both receivers used in the experiments utilize the same 500 kHz signal bandwidth. In the evaluation, we use additive white Gaussian noise (AWGN) to adjust SNR of the transmitted signals from the USRP. The noise signals are applied across the entire signal bandwidth.

Figure 10 plots the achieved BER of the conventional and passive DSSS receivers. First, the conventional (1 channel) case gives the measured result when we only use one of the two receiving branches (we obtain the data from one of the inputs of the XOR gate, see Fig. 7). Then, the conventional (2 channels) case gives the results when both two receiving channels are used for the conventional receiver, in which the gateway transmits continuous waves (CW) on the other channel so we can use the same circuit of passive DSSS to demodulate the data. We observe that there is no obvious
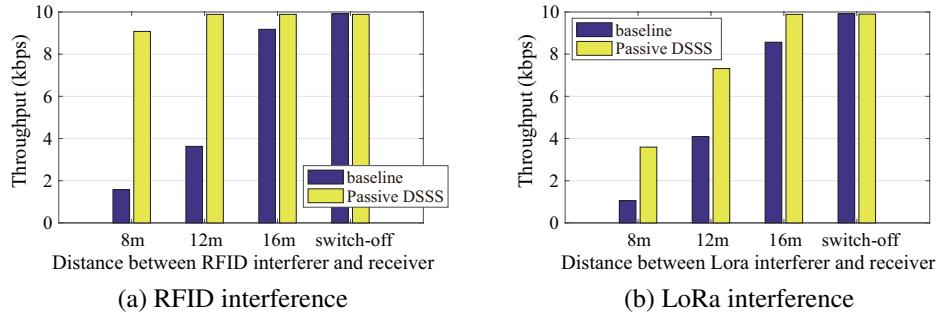
(a) RFID interference   (b) LoRa interference

Figure 11: The measured backscatter downlink throughput achieved when RFID or LoRa interference exists. "switch-off" means no interference. We adopt the 20 dB gain for passive DSSS.

BER difference between the single channel and dual channel cases for the conventional receiver.

For passive DSSS, we first see the result of passive DSSS with the 0 dB processing gain, which uses the two receiving channels but no spread-spectrum. The processing gain refers to the ratio of the spread bandwidth to the baseband bandwidth (*i.e.*, 0 dB represents the ratio is 1). Although the performance of passive DSSS (0 dB) has a slight improvement when SNR is below 15 dB, its BER can be higher than the conventional cases especially when SNR is high. This is mainly due to the signal mismatch between the two channels. Specifically, the expectation of ideally synchronized two channels is impractical due to propagation delay on the RF chain of each receiving channel. Next, we maintain the bandwidth and increase the processing gain to 10 dB. We can see that the BER performance is significantly improved. Finally, we also show the BER result of passive DSSS when the gain is increased to 20 dB. As we expected, the higher processing gain leads to improved BER performance because a higher gain can distribute noise into a wider band during the despreading process, thus resulting in a lower noise power spectral density (PSD).

Moreover, we define the effective communication SNR which is the SNR condition that can suppress the BER to below the threshold of 10^{-2}. With 10 dB processing gain passive DSSS is able to obtain an effective SNR improvement of 6 dB, and with 20 dB processing gain it can obtain 12.8 dB improvement. The primary cost is due to the inadequate performance of the simple envelope detector which is inherently prone to noise. Nevertheless, passive DSSS improves on top of the limits of the conventional receivers, and shows better and more controllable performance with the same coarse envelope detector.

In our following evaluation, we choose to use the conventional (2 channels) setting as the *baseline* to demonstrate the comparative advantage of passive DSSS.

6.3 Real World Evaluation

6.3.1 Realistic Interference Signals

We evaluate the communication performance of passive DSSS with realistic interference from real world. We consider the interference signals from RFID and LoRa transmissions. We employ RFID reader Impinj R420 and LoRa transceiver E32-915T30S from EBYTE as interference sources in the experiment, both of which are configured with 30dBm Tx power. Further, we configure the USRP to transmit downlink data to the receivers, where the passive DSSS transmission has a 20 dB processing gain. We fix the distance between the USRP and receivers at 4 m and vary the distance of the interference source (RFID reader or LoRa transceiver) from 8 to 16 meters away from the receivers. Finally, we switch off the interference source and measure the downlink throughput as a reference.

Figure 11 shows the downlink throughput measurement results under the RFID and LoRa interference. Compared with the baseline approach, passive DSSS can effectively improve the throughput with both RFID and LoRa interference. Further, we can see that passive DSSS can better suppress the interference from RFID than LoRa, mainly because DSSS is inherently suitable for suppressing narrowband interference as the spreading code can distribute the power of the interference signal to a wider band. The RFID interference signals (typical spreading over 100 kHz) are narrower than those from LoRa (250–500 kHz).

6.3.2 Communication Range

We evaluate the communication range of the passive DSSS receivers in both line-of-sight (LOS) and non-LOS (NLOS) scenarios. In the experiments, we employ a low-power amplifier TLV9001 after the envelope detection to achieve a lower receiving threshold for weaker signal cases. The amplifier does not increase the SINR of the received signals as it also amplifies noise and interference.

Figure 12 plots the experimental results with LOS. The LOS experiment considers an outdoor street and the USRP
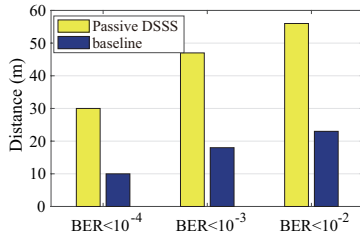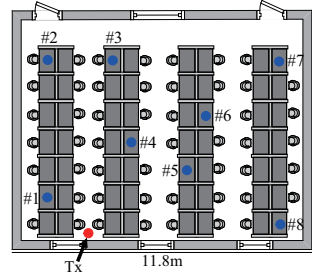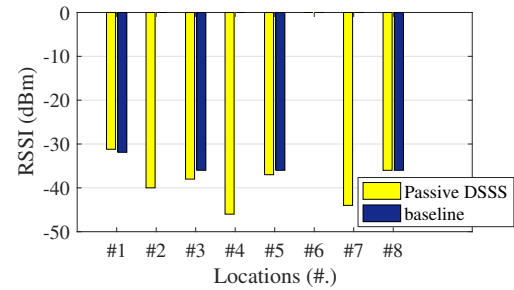
Figure 12: LOS communication range.

(a) NLOS experiment locations

(b) Measured RSS when the BER is below 10^{-2}

Figure 13: NLOS communication range.

conveys test bits to the receiver prototypes with 20 dBm transmission power per channel. We incrementally move the receivers away from the USRP transmitter with 1m step length. At each distance, we record the bits received by each receiver to derive the BER. According to different BER thresholds, we can identify three levels of the effective communication range, *i.e.*, excellent (BER<10^{-4}), good (BER<10^{-3}) and fair (BER<10^{-2}). The figure demonstrates that passive DSSS generally extends the communication range by $\sim$2–3x when compared with the baseline. In particular, the gain is 3x for those communications at the level of "excellent". This is reasonable because passive DSSS uses a wider band to convey the baseband, thus suffering from less noise and interference effects in real environments.

Figure 13 plots the results from the NLOS experiment, where we conduct the experiment in an indoor office. We fix the gateway with 20 dBm transmission power and vary the receivers across 8 different locations within the space as shown in Fig. 13(a). At each location, we measure the BER of the received data for all of the receivers. If the BER is below 10^{-2}, we measure the received signal power (RSS) at the location. We see that passive DSSS works with more locations than the baseline, and discuss the results as follows:

- At locations #2, #4 and #7, only passive DSSS can work. Passive DSSS receiver works with transmissions of RSS as low as -46 dBm owing to its anti-noise capability, while the baseline can only work when the RSS is higher than -36 dBm. As a result, passive DSSS works for nearly all measured locations while the baseline cannot work for half of these locations.

- At location #6, neither passive DSSS nor baseline can work. This location suffers from significant deep fading due to multi-path destructive interference, where both passive DSSS and the baseline do not work.

6.4 Stress-test

To evaluate the performance boundary of passive DSSS, we perform stress-tests with the prototype by manually imposing

interference and noise to the channels until the communication corrupts. We employ a USRP as the source of interference.

6.4.1 Anti-Interference

In this section we evaluate the performance under interference of different modulated signals. We consider RFID signals as an example of amplitude modulation, LoRa signals as frequency modulation and Wi-Fi signals as phase modulation[1]. To this goal, we employ the USRP to measure real transmissions from the above systems correspondingly and replay the measured signals to interfere with passive DSSS (20 dB processing gain). Figure 14 shows the measured BER of passive DSSS with the presence of the three interference signals, respectively. We have the following three observations:

- Although the rationale of envelope detection is based on the amplitudes of the signal envelopes, various types of interference signals may lead to bit errors, because the carriers have different phases compared to the downlink transmission. As a result, although LoRa and Wi-Fi are not based on amplitude modulation, the received downlink transmission envelopes are still destructed when interfered by those signals due to destructive superposition of the signal phases.

- The conventional receiver is more prone to the interference of the RFID and LoRa transmissions than the Wi-Fi transmissions because those are narrowband interference signals (RFID over 100 kHz and LoRa over 250–500 kHz) which have higher PSD. In contrast, the Wi-Fi interference has lower PSD due to the wider transmission bandwidth (22 MHz) and its use of data whitening to distribute the power evenly within the band.

- LoRa transmissions lead to the strongest interference. We observe that RFID adopts amplitude modulation with PIE encoding, and its interference is weak during the

---

[1]We switch the setting to the 2.4GHz band for testing the interference from Wi-Fi signals.
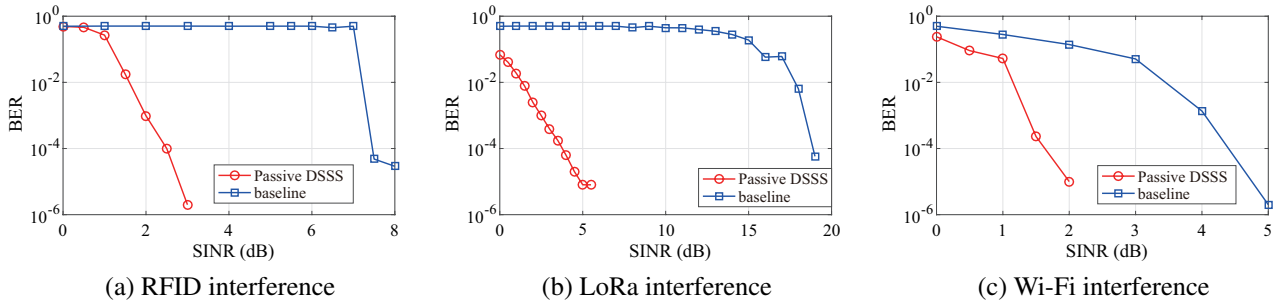
Figure 14: Stress test of interference from RFID, LoRa and Wi-Fi signals.

logic low in PIE symbols. LoRa signals use frequency modulation which does not vary its amplitude, and leads to persistent interference.

In general, compared with the baseline, passive DSSS provides ~5dB, ~13-15dB, and ~3dB gain for RFID, LoRa, and Wi-Fi interference, respectively.

6.4.2 Interference Correlation

We evaluate the performance of passive DSSS with interference of different correlation coefficients between its two communication channels. In the evaluation, we add the same interference signal to both channels and apply AWGN to vary the correlation coefficient between the two channels. Due to the high anti-interference performance, the passive DSSS BER is negligibly low when the SINR is above 3 dB (see Fig. 14). We thus conduct the experiment with transmissions of SINR = 3 dB and examine how the BERs vary when the correlation coefficient across the two channels increases. Figure 15 plots the measured BER of passive DSSS with the RFID, LoRa and Wi-Fi interference. We see that the BER increases when the interference signals on the two channels are more correlated, which is as expected because the despreading quality of passive DSSS depends on the analog integrator to remove the product of the interference compositions on the two channels (Eq. 3). The performance degrades more significantly with the RFID interference than LoRa and Wi-Fi because the ASK signals may directly destruct the envelope of passive DSSS transmissions in both channels.

6.5 Case Study

In this section, we demonstrate the benefit of passive DSSS to practical deployment of backscatter systems in a case study with two different types of deployment.

Monostatic deployment involves a backscatter gateway that comprises the collocated Tx and Rx (Fig. 16(a)), where the gateway transmits to the backscatter device on the downlink and receives the backscattered transmissions on the uplink. In such a case, the key constraint is that the downlink range limits the coverage of the system. Passive DSSS can

effectively improve the downlink range and increase the coverage. To test this, we incorporate a backscatter uplink with 1 kbps chirp modulation. When using the conventional receiver, the coverage of the backscatter system is limited to the downlink range of 26 m. Passive DSSS improves the coverage to 52 m. We observe that the uplink experiences good performance within the extended coverage. The current gateway Rx sensitivity is -97 dBm, which can be further improved to below -130 dBm [15, 24, 49], which suggests more room of improvement with further extended downlink range.

Bistatic/Multistatic deployment is used for long range backscatter communication, where the Tx and Rx gateways are separated (Fig. 16(b)). In practice, multiple Tx gateways are often needed to interrogate the geographically distributed backscatter devices. The short downlink range leads to small coverage of each Tx gateway and as a result more Tx gateways to cover the deployment area. The increased number of gateways may further incur coordination problems among those gateways and lead to higher deployment costs. Passive DSSS mitigates the problem by improving the downlink range. In our experiment, passive DSSS receivers enable the Tx gateway to achieve 4x the coverage area than using the conventional receivers. The uplink distance can arrive at ~108 m when the backscatter devices are located at the edge of the downlink range.

7 Discussion

Bandwidth Usage. As passive DSSS needs two individual channels to transmit the DSSS signals, the bandwidth usage is doubled. Such an issue however is not significant since the downlink to the backscatter devices is typically for control purpose and thus the required bandwidth is small. Our prototype uses 500 kHz for each channel, which is comparable to the bandwidth usage of other IoT communication techniques like LoRa. In addition to that, the gateway can dynamically adjust the bandwidth usage by varying the processing gain in passive DSSS, *e.g.*, the gateway may increase the processing gain to improve interference resilience when detecting interference signals, or reduce the gain for saving the bandwidth usage when the downlink experiences low interference.
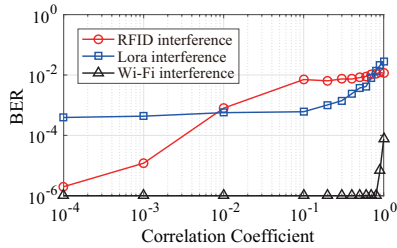
Figure 15: Measured BER with different correlation coefficients between the two passive DSSS channels (SINR=3dB).

Figure 16: Application case study for passive DSSS.

Support for Concurrent Transmissions. While some uses of DSSS achieve concurrent transmissions with the CDMA method, the current passive DSSS design does not support concurrent transmissions because the spreading code is entirely generated by the gateway, which means that all the receivers share the same spreading code. As passive DSSS only requires small bandwidth for each receiving channel, future research may consider exploring frequency multiplexing schemes like frequency-hopping spread spectrum (FHSS) or frequency division multiple access (FDMA) to support concurrent transmissions for passive DSSS.

Signal Jamming. We expect the proposed passive DSSS to make the downlink of backscatter systems resilient to realistic RF interference in practical IoT scenarios. As being discussed in § 4, the interference signals in the two channels are independent and can thus be suppressed by computing the interference correlation. However, if the interference comes from an intentional jamming source, where malicious attackers send a pair of highly correlated interference signals into the two channels, passive DSSS may be compromised. Similarly, high-power interference may impair the SINR over the entire frequency band and thus throttle the communications. We leave the countermeasures to such malicious attacks to future works.

8 Related Work

Passive Radio. There are tremendous existing works which study the backscatter uplink of passive radios including the research on improving the backscatter data rate [51, 52], throughput [16, 19, 20, 65–67], range [43, 49, 53, 54, 56, 65], robustness [33, 59]. Another compelling direction of research explores the inter-operation between backscatter communication and existing wireless systems such as Wi-Fi [4, 18, 25, 26, 64, 68], BLE [11, 18], Zigbee [29, 44], LoRa [40, 43, 49], FM [57] and LTE [5].

On the other hand, there are also a few works to explore other out-of-band wireless channels to convey downlink data to backscatter devices, including the use of the presence and absence of Wi-Fi packets [25], the lengths of Wi-Fi transmis-
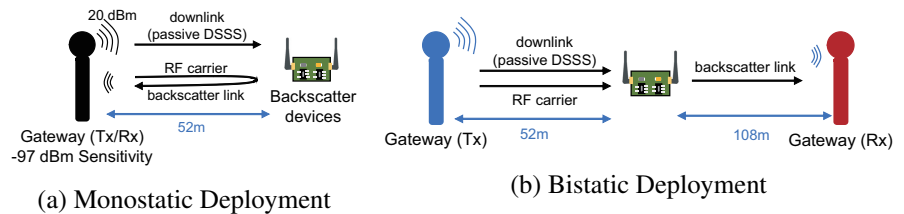
sions [64], reverse engineering of OFDM [18], perturbations to ambient signals [22], and backscatter signals from other tags [31, 32, 41]. In this paper, the proposed passive DSSS presents a direct in-band solution to the downlink problem and is able to support general backscatter communication systems.

Functionality Offloading. The essential idea of backscatter communication is one type of offloading techniques which saves the power consuming RF oscillator in the uplink communication. Recently, a series of works have made great efforts to shift power-starving functionalities from backscatter devices to the gateway side, including storage [48], computation [67], digitization [39, 45], subcarrier generation [47] and sensor control [28]. Passive DSSS by nature belongs to such functionality offloading efforts and shifts the spread-spectrum synchronization to the gateway.

9 Conclusion

This paper proposes passive DSSS to empower downlink transmissions with interference and noise resilience for backscatter communication systems. The proposed design exploits interference suppression across two separate wireless channels to achieve ultra-low power demodulation of spectrum spreading signals. The experimental evaluation with real world interference demonstrates the effectiveness of passive DSSS. We envision that the design of passive DSSS opens a door to making passive communication more practical to future wide area IoT systems and applications.

Acknowledgments

References

[1] Zhenlin An, Qiongzheng Lin, and Lei Yang. Cross-frequency communication: Near-field identification of uhf rfids with wifi! In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, MobiCom '18, pages 623–638, New York, NY, USA, 2018. Association for Computing Machinery.

[2] Holger Arthaber, Thomas Faseth, and Florian Galler. Spread-spectrum based ranging of passive uhf epc rfid tags. *IEEE Communications Letters*, 19(10):1734–1737, 2015.

[3] Venkat Arun and Hari Balakrishnan. Rfocus: Beamforming using thousands of passive antennas. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 1047–1061, Santa Clara, CA, February 2020. USENIX Association.

[4] Dinesh Bharadia, Kiran Raj Joshi, Manikanta Kotaru, and Sachin Katti. Backfi: High throughput wifi backscatter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 283–296, New York, NY, USA, 2015. ACM.

[5] Zicheng Chi, Xin Liu, Wei Wang, Yao Yao, and Ting Zhu. Leveraging ambient lte traffic for ubiquitous passive communication. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 172–185, 2020.

[6] Charles Chien, Igor Elgorriaga, and Charles McConaghy. Low-power direct-sequence spread-spectrum modem architecture for distributed wireless sensor networks. In *ISLPED'01: Proceedings of the 2001 International Symposium on Low Power Electronics and Design (IEEE Cat. No. 01TH8581)*, pages 251–254. IEEE, 2001.

[7] Federal Communications Commission. Regulatory status for using rfid in the epc gen2 (860 to 960 mhz) band of the uhf spectrum. https://www.gs1.org/docs/epc/uhf_regulations.pdf, 2021.

[8] S. Cui, K. C. Teh, K. H. Li, Y. L. Guan, and C. L. Law. Narrowband interference suppression in transmitted reference uwb systems with inter-pulse interference. In *2007 IEEE International Conference on Ultra-Wideband*, pages 895–898, 2007.

[9] Shan Cui, Kah Chan Teh, Kwok H Li, Yong Liang Guan, and Choi Look Law. Performance analysis of transmitted-reference uwb systems with narrowband interference suppression. *Wireless Communications and Mobile Computing*, 9(8):1081–1088, 2009.

[10] F. Dowla, F. Nekoogar, and A. Spiridon. Interference mitigation in transmitted-reference ultra-wideband (uwb) receivers. In *IEEE Antennas and Propagation Society Symposium, 2004.*, volume 2, pages 1307–1310 Vol.2, 2004.

[11] Joshua F Ensworth and Matthew S Reynolds. Every smart phone is a backscatter reader: Modulated backscatter compatibility with bluetooth 4.0 low energy (ble) devices. In *2015 IEEE International Conference on RFID (RFID)*, pages 78–85, April 2015.

[12] Kai Geissdoerfer and Marco Zimmerling. Bootstrapping battery-free wireless networks: Efficient neighbor discovery and synchronization in the face of intermittency. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 439–455. USENIX Association, April 2021.

[13] Jafar Ghaisari and Arash Ferdosi. A direct sequence spread spectrum code acquisition circuit for wireless sensor networks. *International Journal of Electronics*, 98(6):793–800, 2011.

[14] Shyamnath Gollakota, Fadel Adib, Dina Katabi, and Srinivasan Seshan. Clearing the rf smog: Making 802.11n robust to cross-technology interference. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM'11, pages 170–181, New York, NY, USA, 2011. Association for Computing Machinery.

[15] Mehrdad Hessar, Ali Najafi, and Shyamnath Gollakota. Netscatter: Enabling large-scale backscatter networks. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI'19, pages 271–283, USA, 2019. USENIX Association.

[16] Pan Hu, Pengyu Zhang, and Deepak Ganesan. Laissez-faire: Fully asymmetric backscatter communication. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 255–267, New York, NY, USA, 2015. ACM.

[17] Pan Hu, Pengyu Zhang, Mohammad Rostami, and Deepak Ganesan. Braidio: An integrated active-passive radio for mobile devices with asymmetric energy budgets. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM'16, pages 384–397, New York, NY, USA, 2016. Association for Computing Machinery.

[18] Vikram Iyer, Vamsi Talla, Bryce Kellogg, Shyamnath Gollakota, and Joshua Smith. Inter-technology backscatter: Towards internet connectivity for implanted devices. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM'16, pages 356–369, New York, NY, USA, 2016. Association for Computing Machinery.

[19] Meng Jin, Yuan He, Xin Meng, Dingyi Fang, and Xiaojiang Chen. Parallel backscatter in the wild: When burstiness and randomness play with you. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, pages 471–485, 2018.

[20] Meng Jin, Yuan He, Xin Meng, Yilun Zheng, Dingyi Fang, and Xiaojiang Chen. Fliptracer: Practical parallel decoding for backscatter communication. *IEEE/ACM Transactions on Networking*, 27(1):330–343, 2019.

[21] Inyup Kang and Alan N Willson. Low-power viterbi decoder for cdma mobile terminals. *IEEE journal of solid-state circuits*, 33(3):473–482, 1998.

[22] Zerina Kapetanovic, Ali Saffari, Ranveer Chandra, and Joshua R. Smith. Glaze: Overlaying occupied spectrum with downlink iot transmissions. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 3(4), December 2019.

[23] Mohamad Katanbaf, Vivek Jain, and Joshua R. Smith. Relacks: Reliable backscatter communication in indoor environments. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 4(2), June 2020.

[24] Mohamad Katanbaf, Anthony Weinand, and Vamsi Talla. Simplifying backscatter deployment: Full-duplex lora backscatter. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 955–972. USENIX Association, April 2021.

[25] Bryce Kellogg, Aaron Parks, Shyamnath Gollakota, Joshua R. Smith, and David Wetherall. Wi-fi backscatter: Internet connectivity for rf-powered devices. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 607–618, New York, NY, USA, 2014. Association for Computing Machinery.

[26] Bryce Kellogg, Vamsi Talla, Shyamnath Gollakota, and Joshua R. Smith. Passive wi-fi: Bringing low power to wi-fi transmissions. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, pages 151–164, USA, 2016. USENIX Association.

[27] John Kimionis, Aggelos Bletsas, and John N Sahalos. Bistatic backscatter radio for tag read-range extension. In *2012 IEEE International Conference on RFID-Technologies and Applications (RFID-TA)*, pages 356–361. IEEE, 2012.

[28] Songfan Li, Chong Zhang, Yihang Song, Hui Zheng, Lu Liu, Li Lu, and Mo Li. Internet-of-microchips: Direct radio-to-bus communication with spi backscatter. In *The 26th Annual International Conference on Mobile Computing and Networking*, MobiCom '20, London,

United Kingdom, 2020. Association for Computing Machinery.

[29] Yan Li, Zicheng Chi, Xin Liu, and Ting Zhu. Passive-zigbee: Enabling zigbee communication in iot networks with 1000x+ less power consumption. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, SenSys '18, pages 159–171, New York, NY, USA, 2018. Association for Computing Machinery.

[30] Zhuqi Li, Yaxiong Xie, Longfei Shangguan, Rotman Ivan Zelaya, Jeremy Gummeson, Wenjun Hu, and Kyle Jamieson. Towards programming the radio environment with large arrays of inexpensive antennas. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 285–300, Boston, MA, February 2019. USENIX Association.

[31] Vincent Liu, Aaron Parks, Vamsi Talla, Shyamnath Gollakota, David Wetherall, and Joshua R. Smith. Ambient backscatter: Wireless communication out of thin air. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 39–50, New York, NY, USA, 2013. Association for Computing Machinery.

[32] Vincent Liu, Vamsi Talla, and Shyamnath Gollakota. Enabling instantaneous feedback with full-duplex backscatter. In *Proceedings of the 20th Annual International Conference on Mobile Computing and Networking*, MobiCom '14, pages 67–78, New York, NY, USA, 2014. ACM.

[33] Xin Liu, Zicheng Chi, Wei Wang, Yao Yao, and Ting Zhu. Vmscatter: A versatile mimo backscatter. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)*, pages 895–909, 2020.

[34] Tao Long and Naresh R Shanbhag. Low-power cdma multiuser receiver architectures. In *1999 IEEE Workshop on Signal Processing Systems. SiPS 99. Design and Implementation (Cat. No. 99TH8461)*, pages 493–502. IEEE, 1999.

[35] Yunfei Ma, Xiaonan Hui, and Edwin C. Kan. 3d real-time indoor localization via broadband nonlinear backscatter in passive devices with centimeter precision. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking*, MobiCom '16, pages 216–229, New York, NY, USA, 2016. Association for Computing Machinery.

[36] AC McCormick, PM Grant, JS Thompson, T Arslan, and AT Erdogan. Low power receiver architectures for multi-carrier cdma. *IEE Proceedings-Circuits, Devices and Systems*, 149(4):227–233, 2002.

---

[37] L.B. Milstein. Interference rejection techniques in spread spectrum communications. *Proceedings of the IEEE*, 76(6):657–671, 1988.

[38] Carlo Mutti and Christian Floerkemeier. Cdma-based rfid systems in dense scenarios: Concepts and challenges. In *2008 IEEE International Conference on RFID*, pages 215–222. IEEE, 2008.

[39] Saman Naderiparizi, Mehrdad Hessar, Vamsi Talla, Shyamnath Gollakota, and Joshua R. Smith. Towards battery-free hd video streaming. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI'18, pages 233–247, USA, 2018. USENIX Association.

[40] Rajalakshmi Nandakumar, Vikram Iyer, and Shyamnath Gollakota. 3d localization for sub-centimeter sized devices. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, SenSys '18, pages 108–119, New York, NY, USA, 2018. Association for Computing Machinery.

[41] Aaron N. Parks, Angli Liu, Shyamnath Gollakota, and Joshua R. Smith. Turbocharging ambient backscatter communication. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 619–630, New York, NY, USA, 2014. Association for Computing Machinery.

[42] Marco Pausini and Gerard J. M. Janssen. Narrowband interference suppression in transmitted reference uwb receivers using sub-band notch filters. In *2006 14th European Signal Processing Conference*, pages 1–5, 2006.

[43] Yao Peng, Longfei Shangguan, Yue Hu, Yujie Qian, Xianshang Lin, Xiaojiang Chen, Dingyi Fang, and Kyle Jamieson. Plora: A passive long-range data network from ambient lora transmissions. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 147–160, New York, NY, USA, 2018. Association for Computing Machinery.

[44] Carlos Pérez-Penichet, Frederik Hermans, Ambuj Varshney, and Thiemo Voigt. Augmenting iot networks with backscatter-enabled passive sensor tags. In *Proceedings of the 3rd Workshop on Hot Topics in Wireless*, pages 23–27. ACM, 2016.

[45] Vaishnavi Ranganathan, Sidhant Gupta, Jonathan Lester, Joshua R. Smith, and Desney Tan. Rf bandaid: A fully-analog and passive wireless interface for wearable sensors. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 2(2), July 2018.

[46] Mohammad Rostami, Jeremy Gummeson, Ali Kiaghadi, and Deepak Ganesan. Polymorphic radios: A new design paradigm for ultra-low power communication. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 446–460, New York, NY, USA, 2018. Association for Computing Machinery.

[47] Mohammad Rostami, Karthik Sundaresan, Eugene Chai, Sampath Rangarajan, and Deepak Ganesan. Redefining passive in backscattering with commodity devices. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, MobiCom'20, New York, NY, USA, 2020. Association for Computing Machinery.

[48] Mastooreh Salajegheh, Shane S Clark, Benjamin Ransford, Kevin Fu, and Ari Juels. Cccp: Secure remote storage for computational rfids. In *USENIX Security Symposium*, pages 215–230, 2009.

[49] Vamsi Talla, Mehrdad Hessar, Bryce Kellogg, Ali Najafi, Joshua R. Smith, and Shyamnath Gollakota. Lora backscatter: Enabling the vision of ubiquitous connectivity. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 1(3), September 2017.

[50] Vamsi Talla, Joshua Smith, and Shyamnath Gollakota. Advances and open problems in backscatter networking. *GetMobile: Mobile Comp. and Comm.*, 24(4):32–38, March 2021.

[51] Stewart Thomas and Matthew S Reynolds. Qam backscatter for passive uhf rfid tags. In *2010 IEEE International Conference on RFID (IEEE RFID 2010)*, pages 210–214. IEEE, 2010.

[52] Stewart J Thomas and Matthew S Reynolds. A 96 mbit/sec, 15.5 pj/bit 16-qam modulator for uhf backscatter communication. In *2012 IEEE International Conference on RFID (RFID)*, pages 185–190. IEEE, 2012.

[53] Ambuj Varshney, Oliver Harms, Carlos Pérez-Penichet, Christian Rohner, Frederik Hermans, and Thiemo Voigt. Lorea: A backscatter architecture that achieves a long communication range. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, pages 1–14, 2017.

[54] Ambuj Varshney, Carlos Pérez Penichet, Christian Rohner, and Thiemo Voigt. Towards wide-area backscatter networks. In *Proceedings of the 4th ACM Workshop on Hot Topics in Wireless*, pages 49–53, 2017.

[55] Ambuj Varshney, Andreas Soleiman, and Thiemo Voigt. Tunnelscatter: Low power communication for sensor

tags using tunnel diodes. In *The 25th Annual International Conference on Mobile Computing and Networking*, MobiCom '19, New York, NY, USA, 2019. Association for Computing Machinery.

[56] Georgios Vougioukas, Spyridon-Nektarios Daskalakis, and Aggelos Bletsas. Could battery-less scatter radio tags achieve 270-meter range? In *2016 IEEE Wireless Power Transfer Conference (WPTC)*, pages 1–3. IEEE, 2016.

[57] Anran Wang, Vikram Iyer, Vamsi Talla, Joshua R. Smith, and Shyamnath Gollakota. Fm backscatter: Enabling connected cities and smart fabrics. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, pages 243–258, USA, 2017. USENIX Association.

[58] Jingxian Wang, Junbo Zhang, Rajarshi Saha, Haojian Jin, and Swarun Kumar. Pushing the range limits of commercial passive rfids. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 301–316, Boston, MA, February 2019. USENIX Association.

[59] Jue Wang, Haitham Hassanieh, Dina Katabi, and Piotr Indyk. Efficient and reliable low-power backscatter networks. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 61–72, New York, NY, USA, 2012. Association for Computing Machinery.

[60] Po-Han Peter Wang, Chi Zhang, Hongsen Yang, Manideep Dunna, Dinesh Bharadia, and Patrick P. Mercier. A low-power backscatter modulation system communicating across tens of meters with standards-compliant wi-fi transceivers. *IEEE Journal of Solid-State Circuits*, 55(11):2959–2969, 2020.

[61] Die Wu, Muhammad Jawad Hussain, Songfan Li, and Li Lu. R2: Over-the-air reprogramming on computational rfids. In *2016 IEEE International Conference on RFID (RFID)*, pages 1–8, 2016.

[62] Lih-Chyau Wuu, Yen-Ju Chen, Chi-Hsiang Hung, and Wen-Chung Kuo. Zero-collision rfid tags identification based on cdma. In *2009 Fifth International Conference on Information Assurance and Security*, volume 1, pages 513–516. IEEE, 2009.

[63] Zhengyuan Xu, B.M. Sadler, and Jin Tang. Data detection for uwb transmitted reference systems with inter-pulse interference. In *Proceedings. (ICASSP '05). IEEE International Conference on Acoustics, Speech, and Signal Processing, 2005.*, volume 3, pages iii/601–iii/604 Vol. 3, 2005.

[64] Pengyu Zhang, Dinesh Bharadia, Kiran Joshi, and Sachin Katti. Hitchhike: Practical backscatter using commodity wifi. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*, SenSys '16, pages 259–271, New York, NY, USA, 2016. Association for Computing Machinery.

[65] Pengyu Zhang and Deepak Ganesan. Enabling bit-by-bit backscatter communication in severe energy harvesting environments. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 345–357, USA, 2014. USENIX Association.

[66] Pengyu Zhang, Jeremy Gummeson, and Deepak Ganesan. Blink: A high throughput link layer for backscatter communication. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, pages 99–112, New York, NY, USA, 2012. ACM.

[67] Pengyu Zhang, Pan Hu, Vijay Pasikanti, and Deepak Ganesan. Ekhonet: High speed ultra low-power backscatter for next generation sensors. In *Proceedings of the 20th annual international conference on Mobile computing and networking*, pages 557–568. ACM, 2014.

[68] Renjie Zhao, Fengyuan Zhu, Yuda Feng, Siyuan Peng, Xiaohua Tian, Hui Yu, and Xinbing Wang. Ofdma-enabled wi-fi backscatter. In *The 25th Annual International Conference on Mobile Computing and Networking*, MobiCom '19, pages 20:1–20:15, New York, NY, USA, 2019. ACM.

Check-N-Run: a Checkpointing System for Training Deep Learning Recommendation Models

Assaf Eisenman[1], Kiran Kumar Matam[1], Steven Ingram[1], Dheevatsa Mudigere[1], Raghuraman Krishnamoorthi[1], Krishnakumar Nair[1], Misha Smelyanskiy[1], and Murali Annavaram[1,2]

[1]Facebook, Inc, [2]USC

Abstract

Checkpoints play an important role in training long running machine learning (ML) models. Checkpoints take a snapshot of an ML model and store it in a non-volatile memory so that they can be used to recover from failures to ensure rapid training progress. In addition, they are used for online training to improve inference prediction accuracy with continuous learning. Given the large and ever-increasing model sizes, checkpoint frequency is often bottlenecked by the storage write bandwidth and capacity. When checkpoints are maintained on remote storage, as is the case with many industrial settings, they are also bottlenecked by network bandwidth. We present Check-N-Run, a scalable checkpointing system for training large ML models at Facebook. While Check-N-Run is applicable to long running ML jobs, we focus on checkpointing recommendation models which are currently the largest ML models with Terabytes of model size. Check-N-Run uses two primary techniques to address the size and bandwidth challenges. First, it applies differential checkpointing, which tracks and checkpoints the modified part of the model. Differential checkpointing is particularly valuable in the context of recommendation models where only a fraction of the model (stored as embedding tables) is updated on each iteration. Second, Check-N-Run leverages quantization techniques to significantly reduce the checkpoint size, without degrading training accuracy. These techniques allow Check-N-Run to reduce the required write bandwidth by 6-17× and the required capacity by 2.5-8× on real-world models at Facebook, and thereby significantly improve checkpoint capabilities while reducing the total cost of ownership.

1 Introduction

Deep learning has become extensively adopted in many production scale data center services. In particular, deep learning enabled recommendation systems power a wide variety of products and services. These include e-commerce marketplaces (e.g. Amazon, Alibaba) for recommending items to purchase [30, 33], social media platforms (e.g. Facebook, Twitter) for providing the most relevant content [14], entertainment services (e.g. Netflix, Youtube) for promoting new playlists [7, 12], and storage services (e.g. Google Drive) for enabling quick access to stored objects [4].

At Facebook's datacenter fleet, for example, deep recommendation models consume more than 80% of the machine learning inference cycles and more than 50% of the training cycles. Similar demands can be found at other companies [16].

Typically, the accuracy of deep learning algorithms increases as a function of the model size and number of features. For instance, the recommendation model size at Facebook grew more than 3× in under two years (see Figure 4). Recommendation models are particularly in need of massive model size to store sparse model features. Hence, they are orders of magnitude larger than even the largest DNNs, such as Transformer based models [32], and often occupy many terabytes of memory per model [38]. Because of their large size, these models also must be trained with massive datasets and run in a distributed fashion. Therefore, training recommendation models at production scale may take several days, even when training on highly optimized GPU clusters.

Given that the training runs span multiple GPU clusters over multiple days and weeks, there is an abundance of failures that a training run may encounter. These include network issues, hardware failures, system failures (e.g. out of memory), power outages, and code issues. Checkpointing is an important functionality to quickly recover from such failures for reducing the overall training time and ensure progress. Checkpoints are essentially snapshots of the running job state taken at regular intervals and stored in persistent storage. To recover from failure and resume training, the most recent checkpoint is loaded.

In addition to failure recovery, checkpoints are needed for moving training processes across different nodes or clusters. This shift may be required in cases such as server maintenance (e.g. critical security patches that could not be postponed), hardware failures, network issues, and resource optimization/re-allocation. Another important use-case of checkpoints is publishing snapshots of trained models in real time to improve inference accuracy (online training). For instance, an interim model can be used for prediction serving (obtained by checkpointing), while the model is still being trained over more recent dataset for keeping the inference model freshness. Checkpoints are also used for performing transfer learning, where an intermediate model state is used as a seed, which is then trained for a different goal [26].

Checkpoints must meet several key criteria:

(1) **Accuracy:** They must be accurate to avoid training accuracy degradation. In other words, when a training run is restarted from a checkpoint, there should be no perceivable

difference in the training accuracy or any other related metric. As has been stated in prior works on production scale recommendation models [38], even a tiny decrease of prediction accuracy would result in an unacceptable loss in user engagement and revenues. Hence, preserving accuracy is a constraint for checkpoint management in recommendation models.

(2) **Frequency:** Checkpoints need to be frequent for minimizing the re-training time (the gap between failure time and the most recent checkpoint timestamp) after resuming from a checkpoint. For instance, taking a checkpoint every 1000 batches of training data may lead to wasting time re-training those 1000 batches. Taking a checkpoint after 5000 batches leads to $5\times$ more wasted work in the worst case. In the case of online training, the checkpoint frequency directly impacts how quickly the inference adapts in real time and its prediction accuracy.

(3) **Write Bandwidth:** Checkpoints at Facebook, as well as in other industrial settings, are written to remote storage to provide high availability (including replications) and scalable infrastructure. Writing multiple large checkpoints concurrently from different models that are being trained in parallel (e.g., thousands of checkpoints, each in the order of terabytes) to remote storage requires substantial network and storage bandwidths, which constitute a bottleneck and limit the checkpoint frequency. Hence, it is necessary to minimize the required bandwidth to enable frequent checkpoints.

(4) **Storage capacity:** Storing checkpoints at-scale requires hundreds of petabytes of storage capacity, with high-availability and short access times. Checkpoints at Facebook are typically stored for many days, thus the number of stored checkpoints at a given time is reflected by the number of training jobs in that time period. While the last checkpoint per run is saved by default, it is often useful to keep several recent checkpoints (e.g. for debugging and transfer learning). As models keep getting larger and more complex, resulting in an ever increasing storage capacity demand, it is necessary to reduce the corresponding checkpoint size to minimize the required storage capacity for accommodating all checkpoints.

Unfortunately, standard compression algorithms such as Zstandard [6] are not useful enough for deep recommendation workloads. In our experience, we were able to reduce the checkpoint size and the associated write-bandwidth and storage capacity by at most 7% using Zstandard compression.

Based on the above challenges, we present Check-N-Run, a high-performance scalable checkpointing system, particularly tailored for recommendation systems. Check-N-Run's main goal is to significantly reduce the required write-bandwidth and storage capacity, without degrading accuracy. Our goal is to work within the accuracy degradation constraint set by business needs ($< 0.01\%$).

Recently, CheckFreq has demonstrated the benefits of checkpointing for deep neural networks(DNNs) [19]. Check-Freq proposed *adaptive rate tuning* to dynamically determine when to initiate a checkpoint, and a *two-phase* strategy to enable checkpoint storage and training to move concurrently. However, recommendation models provide unique opportunities to tackle checkpointing challenges that are not afforded in traditional DNNs. First, recommendation models update only part of the state after every batch. Hence,it is possible to explore checkpointing strategies that can incrementally store the checkpoint. Second, recommendation model sizes can exceed Terabytes, which stress even planetary scale storage systems. Check-N-Run builds on several techniques:

(1) **Differential checkpointing:** Check-N-Run utilizes differential checkpointing for reducing the checkpoint write bandwidth. This is a technique that is particularly well suited for recommendation models where only a small fraction of the model parameters are updated after each iteration. This is a unique property of recommendation models. In traditional DNNs the entire model is updated after each iteration since gradients are computed for all the model parameters. Recommendation models, on the other hand, access and update only a small fraction of the model during each iteration. Differential checkpoints leverage this observation by tracking and storing the modified parts of the models.

(2) **Quantization:** Check-N-Run leverages quantization techniques to significantly reduce the size of checkpoints. This optimization reduces the required write bandwidth to remote storage, and the storage capacity. While quantization of model parameters during training may have a negative impact on accuracy, checkpointing has the advantage that quantization is only used to store the checkpoint, while full precision is used for training. The only time checkpoint quantization may impact training accuracy is when the quantized checkpoint is restored and de-quantized to resume training. Check-N-Run leverages this insight to maintain training accuracy within our strict bounds.

(3) **Decoupling:** To minimize the run time overhead and training stalls, Check-N-Run creates distributed snapshots of the model in multiple CPU host memories. That way, training on the GPUs can continue while Check-N-Run is optimizing and storing the checkpoints in separate processes running on the CPU in the background. Check-N-Run enables the frequent checkpointing of hundreds of complex production training jobs running in parallel over thousands of GPUs, each job training a very large model (in the order of terabytes). This decoupling approach is also proposed in CheckFreq which separates *snapshot* process from the *persist* storage process [19]. Our implementation of decoupled checkpointing leads to less than 0.4% of time when the trainer processes must pause to take a snapshot. Hence, the impact of taking a checkpoint on the *training speed is negligible*.

The contributions in this paper include:
(1) To our knowledge, Check-N-Run is the first published checkpointing system that uses quantization and differential views for recommendation systems at-scale, demonstrated on real-world workloads.
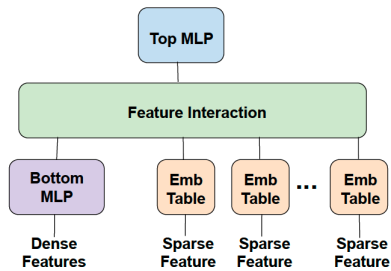(2) We design and evaluate a wide range of checkpoint quanti-

Figure 1: A typical recommendation model. It consists of large embedding tables for mapping the sparse features to vectors, and MLPs for processing the dense features (bottom MLP). These feature interactions are combined in the top MLP. The interaction op combines the dense and sparse features, in order to train with them together.

zation approaches to significantly reduce the checkpoint size by 4-13×, without degrading the training accuracy.

(3) We introduce differential checkpoints, which store the modified part of the model, rather than storing the entire model. Differential checkpoints reduce the average write bandwidth by more than 50%, with no impact on accuracy.

(4) Finally, we demonstrate a heterogeneous checkpointing mechanism that combines differential checkpointing with quantization. Check-N-Run provides $6 - 17\times$ improvement in the required checkpointing write bandwidth, and $2.5 \times - 8\times$ less capacity, without sacrificing accuracy and run time.

2 Background

2.1 Recommendation Models

Recommendation models are a variant of deep learning models that are used to provide recommendations to users based on their past interactions with a digital service. Recommendation systems are often used in commercial settings and dominate the datacenter capacity for AI training and inference [22]. Broadly speaking, recommendation models use a combination of a fully connected multi-layer perceptron (MLP) to capture the dense features, and a set of sparse features that capture categorical data such as a user's past activity and main characteristics of a post. Figure 1 depicts a typical recommendation model used in this study.

Sparse features are captured through embedding tables [10], which map each category to a dense representation in an abstract space. Each embedding table may contain many millions of vectors, with different vector dimensions (e.g. 64), where each element is a 32-bit floating-point number during training. Embedding tables constitute the majority of the model footprint, and account for $> 99\%$ of its size. A training sample includes a set of vector indices per embedding table, which is used to extract the corresponding multi-hot encoded vectors stored in those indices. Once the embedding vectors are extracted, they are trained with a deep neural network.

The size of the sparse layer prevents the use of pure data

parallel training, since it would require replicating the large embedding tables on every device. The large footprint of the sparse layer requires the distribution of the embedding tables across multiple devices, emulating model parallelism. MLP parameters, on the other hand, have a relatively small memory footprint, but they consume a lot of compute. Hence, data-parallelism is an effective way to enable concurrent processing in the MLPs, by running separate samples on different devices and accumulating the updates. Our training system thus uses a combination of model parallelism for the sparse layer, and data parallelism for the MLPs. This hybrid approach mitigates the memory bottleneck of accessing the embedding tables by distributing these tables across multiple GPUs, while parallelizing the forward and backward propagation over the MLPs.

2.2 High Performance Training at Facebook

Given the prominence of recommendation models in today's social media platforms, these models are trained on dedicated clusters [23, 38]. At Facebook, over 50% of the ML training cycles are dedicated solely to recommendation models. Figure 2 illustrates the training pipeline for deep learning recommendation models. Broadly speaking, it consists of 3 stages, located at separate clusters: dataset reader cluster, training cluster, and remote checkpoint storage.

To support high-performance training, our training system relies on clusters of GPUs attached to host CPUs as shown in Figure 2 (*training cluster*). The GPUs accelerate the training tensor operations and accommodate the model parameters, while CPUs run other tasks, such as data ingestion and checkpoint handling. Each training cluster contains 16 nodes, each with 8 GPUs attached to multi-core CPU. Hence, training a model on an entire cluster would partition the embedding tables and the training batches over 128 GPUs, in addition to replicating the MLPs over these GPUs.

In cases where GPU memory is not sufficient for accommodating the models, our training system leverages hierarchical memory: the model parameters are stored in DRAM, while GPU memory serves as a cache.

The model parameters are updated *synchronously* [3], ensuring the updated parameters across the devices are consistent before each training iteration. This is needed for enabling scalable training and avoiding accuracy degradation when training in high throughput. Fully synchronized training avoids regression in the model quality with increased scale and decouples model quality from training throughput. We employ a decentralized model synchronization approach in which each node performs the computations on its local part of the model. For the data-parallel MLPs, an "AllReduce" communication is done in the backward pass to accumulate the gradients computed on the multiple GPUs (each with a different sub-batch of data). For the model-parallel sparse layer, an "AlltoAll" communication [23] occurs both in the forward
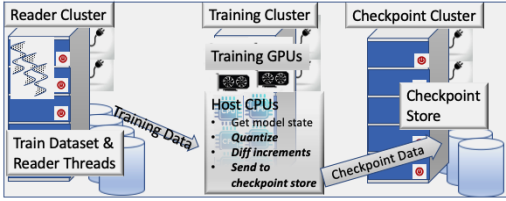
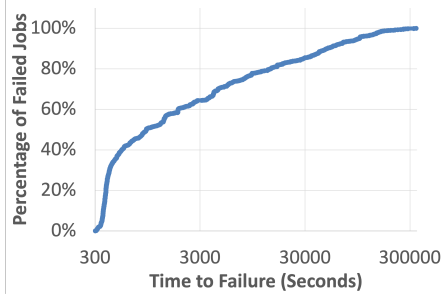Figure 2: An Overview of Training and Checkpoint Systems

Figure 3: Training job failure CDF in our cluster. Jobs that fail within 5 minutes are removed since they are usually simple user setup errors.

pass (to communicate the looked-up embedding vectors), and in the backward pass (to communicate the embedding vector updates). Checkpoint write process is done in the background (using dedicated CPU processes in the trainer nodes), while the training process continues in GPU.

Since the dataset used for training (i.e., training samples) is enormous, and training has to be done at high-throughput (e.g. 500K training queries per second called QPS), it is important to make sure that reading training data will not become a bottleneck. As such, the training system deploys a separate distributed reader tier (shown as *Reader Cluster* in Figure 2), which enables reading resources and training resources to scale independently. Each training cluster uses hundreds of reader nodes residing in a separate cluster, in charge of saturating the trainer with training samples.

Checkpoints of the training job state (consisting of both the reader and trainer states) are stored at a separate, remote storage (shown as *Checkpoint Cluster* in Figure 2).

Training jobs are submitted to this infrastructure through an internally developed job scheduling interface. Schedulers like Bistro [11] and PBS [15] handle job and user priorities, and manage the job queue. The scheduler assigns jobs based on the job configuration and cluster availability. It continuously monitors both the job progress as well as system health status.

3 Motivation

3.1 Training Failures

We analyzed the training job failures on a training system consisting of 21 training clusters, over a one month period. Figure 3 presents the time-to-failure statistics. The X-axis shows the total execution time that was completed by a job before it failed, and the Y-axis shows the percentage of failed jobs which failed before that time. The data shows that longest
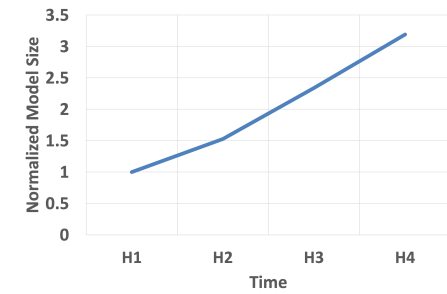

Figure 4: The normalized model size over the past 2 years

10% of the failed jobs ran for at least 13.5 hours before they fail, and the top 1% of the failed jobs fail after executing for not less than 53.9 hours. Note that many of these jobs require 128 GPUs spanning many nodes, that are expensive to maintain and run. These training jobs interact with multiple systems for training. For instance, the training process accesses training samples provided by a separate reader cluster. As such, any one failure in these inter-connected systems will hobble the entire training progress. This data shows the criticality of efficient checkpointing to ensure training progress. Otherwise, long running training jobs may never complete their task. This data motivated the need for Check-N-Run.

As the model sizes are growing continuously, training is getting distributed even more widely across the datacenters. Hence, the failure rates are expected to continue to grow significantly. Thus checkpointing of large model training is a critical problem for any production model.

3.2 Model Size

Recommendation model sizes are often massive due to their large sparse features (represented as embedding tables). Typically, the accuracy of these models increases as a function of the model size. Figure 4 shows our model size increase over the past 2 years (exact model size is confidential). As can be seen, it increased by over $3\times$. Given the large and ever-increasing model sizes, checkpoints are often bottlenecked by write-bandwidth and storage capacity.

3.3 Model Updates

Another set of motivation data shows the sparsity of model updates over time. We analyze one of the largest recommendation models at Facebook and observe that due to large model sizes and their high sparsity, only a fraction of the embedding vectors is modified in a given training interval. Figure 5 shows the percentage of the model that is modified, as a function of the number of training records used to train, starting from three different initial states. The curve starting at the origin shows what fraction of the model size is updated starting from the first training record and ending at about 11 billion training records. As can be seen, even after 11 billion training records, the fraction of the model that is accessed grows slowly and
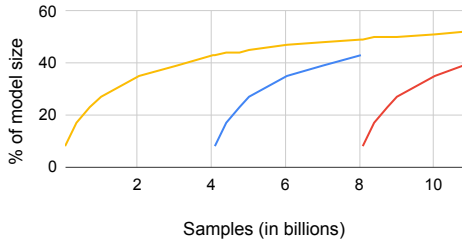
Figure 5: The fraction of model size modified w.r.t. the number of training samples, measured from 3 different starting points
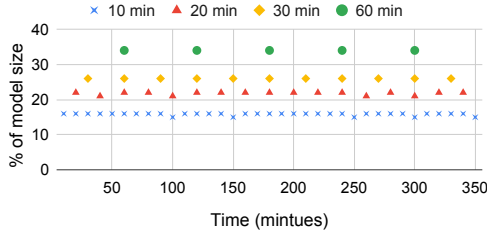

Figure 6: The fraction of model size that is modified during different time intervals

reaches only 52%. Furthermore, the fraction of the model updated during a training interval is expected to continue to shrink as model sizes keep increasing, which is the general industry trend.

The second curve in Figure 5 shows how the fraction of the updated model grows if we only observe updates starting at the 4 billionth training record. The third curve shows the same data starting at about the 8 billionth training record. It is interesting to note that no matter when we start observing the model size growth, the fraction of the modified model size follows a similar slope. This fact is made more clear in Figure 6, which plots the fraction of model size that is modified during a given time interval. For a given interval length, the fraction of model size that is modified remains almost the same in all intervals (e.g., in each 30 minute intervals, about 26% of the model is modified). The above data indicates that at each iteration only a tiny fraction of the model is updated.

4 Check-N-Run Design Overview

Check-N-Run is a distributed checkpointing system for training systems at scale, implemented in our PyTorch training framework. Check-N-Run generates accurate checkpoints of the training system state and ensures there is no accuracy degradation due to creating or loading from a checkpoint. Since training accuracy is a main concern, we are not interested in exploring choices that come at the expense of an unacceptable training accuracy loss, even as small as 0.01%. In this section, we provide an overall overview, while in section 5 we discuss the checkpoint optimization details. Figure 7 illustrates Check-N-Run's overall design, showing what functionality is implemented in each of the reader, trainer and checkpoint storage tiers. Check-N-Run is implemented pri-
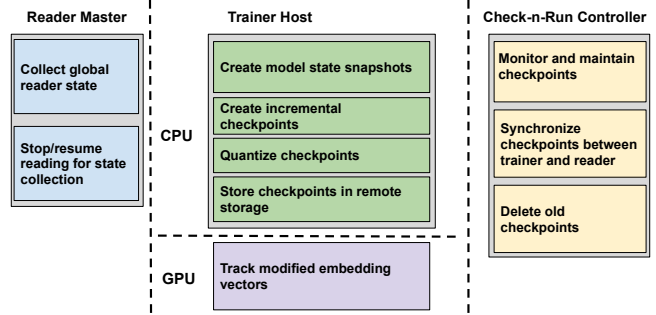

Figure 7: Check-N-Run design components

marily on the host CPU of the training cluster, while its tracking mechanism (described in 5.1.1) is implemented in GPU. It has additional coordination threads running on the reader master (in the reader cluster) and a lightweight Check-N-Run controller that may reside in a dedicated host. Checkpoints are written to remote object storage to provide high availability (including replications) and storage scalability.

4.1 What to Checkpoint?

The trainer state consists of all the model layers (including the sparse and dense features), the optimizer state, and the relevant metrics. Since the MLPs are replicated and maintained with a consistent view during training, it is enough to read them from a single GPU for checkpointing. The embedding tables, however, are distributed across GPUs and hence each GPU must provide a snapshot of the embedding tables that are stored in its local memory.

When a training job resumes from a checkpoint, the run should still train the same training dataset as the original run. Hence, the checkpoint must also include the reader state. This is important, for example, to avoid training the same sample twice. The reader reads the dataset in the granularity of splits (each split represents successive rows of the dataset). Its state includes the set of splits that are pending, and the set of splits that have been partially read (including their cursor position). Note that checkpoints that are intended solely for alternate use-cases such as online training (frequently updating an already trained model running in inference) and transfer learning, do not require the reader state.

Avoiding the trainer-reader state gap: In a production scale training system, checkpointing has unique challenges. As described earlier, a separate set of distributed readers is in charge of feeding the trainers with batches in sufficient throughput. Since readers and trainers work in a distributed fashion in our training system (and reside in separate clusters), many training records are in-flight and reside in different queues. These are batches that have been read by the reader, but have not been consumed by the trainer yet. They constitute a gap between the reader state and the trainer state, which may affect accuracy when loading from a checkpoint. After resuming from a checkpoint, the reader may not know which of

the training samples have been processed. To avoid this gap, Check-N-Run's controller communicates to a *coordination thread* running on the reader master how many batches to read until the next checkpoint. The reader makes sure to read this exact number of batches. For example, if the checkpointing interval is 1000 batches, the reader will provide exactly 1000 batches to the trainer and then stop reading. When trainer finishes processing the 1000th batch and a checkpoint is triggered, there will be no in-flight batches. That way, there is essentially no gap between the reader state and the trainer state. After reader state has been collected, Check-N-Run signals the reader to resume reading the number of batches until the next checkpoint.

4.2 Decoupled Checkpointing

Checkpointing requires the model parameters to be atomically copied for further processing and storage. Note that this atomicity is important for consistency. Otherwise, training processes may update the model during the copying time window, causing substantial consistency challenges and potential accuracy degradation when loading checkpoints. Check-N-Run achieves atomicity by stalling training at the start of a checkpoint and transferring the model state from GPU memory to host CPU memory. Training is stalled only when creating a copy of the model parameters in-memory. As soon as the model snapshot is ready, dedicated CPU processes are in charge of creating, optimizing, and storing checkpoints in the background, while training continues on the GPUs. All training nodes concurrently create a unique snapshot of their own local part of the model. For instance, if the embedding tables are distributed across multiple nodes, each node snapshots its own embedding tables and transfers that information to the host CPU.

Using this approach to create a snapshot scales well with larger models and more nodes, as utilizing additional nodes does not increase the checkpoint performance overhead. For instance, creating a snapshot (in CPU DRAM) of a typical model residing in the GPU memory and partitioned across 16 nodes, each with 8 GPUs (total of 128 GPUs), would stall training in our system for less than 7 seconds. When checkpointing every 30 minutes (our default), stall time would be a negligible fraction ($< 0.4\%$).

4.3 Checkpointing Frequency

The checkpointing frequency is bounded by the available write bandwidth to remote storage. Since Check-N-Run leverages remote storage, it is also limited by available network bandwidth. With larger and ever increasing model sizes, as well as the growing number (e.g. hundreds) of training clusters that concurrently train and checkpoint separate training jobs, these resources constitute a bottleneck. In Check-N-Run, two consecutive checkpoints cannot overlap, and writing of
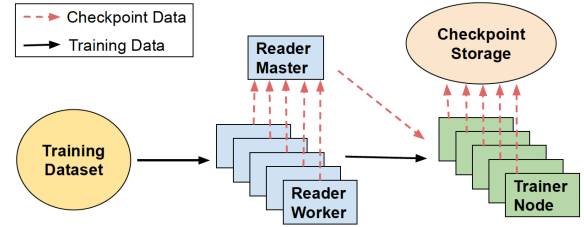

Figure 8: High-level data flow during training

the current checkpoint must be completed or cancelled before a new checkpoint can be created. That way, the current checkpoint can utilize all available resources to minimize the write latency (i.e., the time it takes a checkpoint to become valid and ready to use). Based on our model size and system resource considerations, we initiate a new checkpoint every 30 minutes by default. In section 5 we describe the optimizations leveraged by Check-N-Run to significantly reduce the required resources, providing a scalable solution to enable high frequency checkpointing and reduce the associated total cost of ownership (TCO).

4.4 Check-N-Run Workflow

We define the *checkpoint interval* as the number of trained batches between two consecutive checkpoint. The checkpoint operation is triggered at the end of each checkpoint interval (a configurable number of batches), after the backpropagation stage of the last batch in that interval. Since our training system is fully synchronous, all GPUs will reach their last batch in the checkpoint interval and wait until the next batch is started. The checkpointing process consists of 3 main stages: (1) Create an in-memory snapshot of the training state (2) Build an optimized checkpoint (3) Write the checkpoint to storage.

Figure 8 depicts the high-level data flow between the reader, trainer, and remote checkpoint storage during training.

At the beginning of the training run, Check-N-Run's controller communicates to the coordination thread on the reader master node in the reader tier, to inform what is the checkpoint interval, i.e. how many batches to read until the next checkpoint. The reader master then initiates several reader worker threads which start reading data from the training dataset to provide the trainer nodes. When a checkpoint is triggered, Check-N-Run collects the reader state at this point, which specifies what parts of the training dataset have been read so far. At the same time, all trainer nodes are stalled to concurrently create a snapshot of their local state, by copying the model state from each of their GPUs into host CPU DRAM. As soon as all snapshots are ready, training continues. This decoupling mechanism essentially minimizes the checkpointing process from bottlenecking the trainer.

In step 2, Check-N-Run leverages several techniques to reduce the required checkpoint capacity and write bandwidth, as described in section 5. These techniques are concurrently

applied by each trainer node and run in dedicated CPU processes that are resident on the host CPUs in the trainer tier, outside of the GPU critical path. Only the tracking mechanism described in 5.1.1 is implemented in GPU.

In step 3, the checkpoint is moved to remote checkpointing storage. Note that the optimization process in step 2 works on chunks of embedding vectors at a time. Hence these chunks of quantized and differential checkpoints can be stored in a pipelined manner, enabling concurrent optimization and the checkpoint storing process. When all nodes finish storing their part of the checkpoint successfully, Check-N-Run's controller will declare a new valid checkpoint. At that stage, an older checkpoint may be deleted by the controller (based on the system configuration). Multiple checkpoints can be stored depending on the needs and use cases.

5 Checkpoint Optimizations

5.1 Differential Checkpointing

One-Shot Differential Checkpoint: Motivated by the insight presented in Section 3 regarding the fraction of the model size that is modified after each iteration, we introduce differential checkpoints. Differential checkpointing starts with a single checkpoint taken as a full baseline checkpoint, including all the embedding vectors. From this point, the system starts tracking all modified vectors to create a differential view of the embedding vectors that would have to be included in the next checkpoint. Each differential checkpoint would then store only the vectors that were modified since the baseline checkpoint. To resume from a checkpoint, both the baseline checkpoint and the most recent differential checkpoint have to be read. We refer to this method as *One-shot baseline*.

Consecutive Incremental Checkpoint: We also explored an alternative way, which we denote as *consecutive incremental checkpoint*. This approach stores the vectors that were modified only during the last checkpoint interval, rather than storing the vectors from a baseline checkpoint. This method reduces the current checkpoint size, since only those modified vectors since the last interval are stored. But this approach would require keeping all previous incremental checkpoints for reconstructing the model when resuming from a checkpoint. Note that in our remote object storage, merging consecutive incremental checkpoints would require moving all the data back to the CPU host, which costs substantial bandwidth. Keeping all the incremental checkpoints leads to higher storage capacity since a vector that is modified during multiple intervals will have multiple copies stored. However, incremental checkpoints are useful for use cases such as online training, where checkpoints are directly applied to an already-trained model in inference to improve its freshness and accuracy.

Intermittent Differential Checkpoint: One challenge with the above methods is that the checkpoint size gradually increases. As training progresses, the number of modified model parameters over a baseline will increase. One way to reduce this growth is to checkpoint a full model intermittently, so that the differential view size can be reduced. We exploit the observation from Figure 5 that the modified model size grows similarly from three different starting points.

We use a simple history based predictor to decide when to take a full checkpoint. At the end of each checkpoint interval, it estimates the expected cumulative size of future checkpoints if another differential checkpoint is taken, compared with the total expected size if a full checkpoint is taken (which will then reduce the future checkpoint sizes). Based on this comparison, the system decides whether to take a full checkpoint or stay with a differential checkpoint. The algorithm for this selection is as follows:

Let $S_1, S_2, ..., S_i$ be the sizes of the past i differential checkpoints, which follow a full baseline checkpoint with a size S_0. S is expressed as a fraction of the full baseline checkpoint, such that $S_0 = 1$. Then, at the $(i+1)^{th}$ interval, Check-N-Run faces two options: (1) create a full baseline checkpoint, or (2) create another differential checkpoint. If a full baseline checkpoint is created, we estimate the future cumulative checkpoint size F_c of the next $i+1$ intervals to be similar to the past $i+1$ intervals. That is, $F_c = 1 + S_1 + S_2, ..., +S_i$. Alternatively, if a differential checkpoint is created, the total checkpoint size of the next $i+1$ intervals is larger than, or at best equal to $I_c = (i+1) * S_i$. This relation holds, because starting at interval $i+1$ differential checkpoint size will be at least S_i. Thus, at the $(i+1)^{th}$ interval, we do a full checkpoint if $F_c \leq I_c$, else we do a differential checkpoint. We term this approach as *intermittent differential checkpoint*. This approach can be improved with more accurate prediction models, which are part of future work.

5.1.1 Efficient Tracking

Check-N-Run is intended for high-performance training, hence it aims to minimize the overhead of tracking which embedding vectors are modified. Since the embedding tables are partitioned across the GPUs, each GPU separately tracks the accesses to its local embedding tables. For the sake of simplicity, the training records are tracked during the forward pass, as most of the embedding vectors accessed in the forward pass are also modified during the backward pass. During tracking, each GPU updates a bit-vector associated with its local embedding vectors. This bit-vector is used as a mask to determine which embedding vectors are modified during the training process, and should eventually be included in the next differential checkpoint. Note that the bit-vector memory footprint is low (typically less than 0.05%, on the order of several MBs per GPU).

We utilize idle GPU cycles to reduce tracking overhead, by scheduling the tracking functionality during the "AlltoAll" communication phase (described in section 2.2). Using this implementation, the tracking overhead is reduced to $\approx 1\%$ of

the iteration training time.

5.2 Checkpoint Quantization

The second technique that Check-N-Run uses is quantization of checkpoints. While quantization has been adopted in some cases for reducing model size during inference [18, 37, 40], or to reduce communication costs of parameter aggregation [36], training is typically done in single-precision floating-point format (FP32) to maximize training outcomes and model accuracy. Check-N-Run leverages quantization techniques to significantly reduce the checkpoint size during training, without sacrificing training accuracy.

Quantization in Check-N-Run is decoupled from the training process and is done in background CPU processes after a model snapshot has been created. Hence, it does not affect training throughput. Since quantization is applied to a chunk of rows, the quantized checkpoint store operation does not have to wait until the entire checkpoint is quantized and can store the quantized rows eagerly as needed.

The quantization of embedding tables is usually applied in the granularity of an entire embedding vector. We aim to minimize the error between the original vector $X \in \mathbb{R}^n$ and the quantized vector $Q \in \mathbb{Z}^n$, by minimizing $\|X - Q\|_2$. We define the *mean ℓ_2 error* of an entire quantized checkpoint as: $\frac{1}{m} \sum_{i=0}^{m} \|X_i - Q_i\|_2$, where m is the total number of embedding vectors in the checkpoint. The *mean ℓ_2 error* metric is a good proxy for accuracy loss because the model accuracy is dependent on the values of the embedding tables. This metric captures the distance between the original value of an embedding entry without quantization and the new value produced due to quantization. We observed that this difference provides the first order impact on the accuracy loss and use it to compare different quantization methods. In section 6, we demonstrate how training accuracy is impacted by Check-N-Run's quantization schemes.

In this work we explored 3 quantization methods, *Uniform Quantization, Non-Uniform Quantization and Adaptive Quantization*, to empirically evaluate which approach provides the lowest *mean ℓ_2 error*. Let x be the value of an element in an embedding vector $X \in \mathbb{R}^n$, clipped to the range $[x_{min}, x_{max}]$. N-bits quantization maps x to an integer in the range $[0, 2^N - 1]$, where each integer corresponds to a quantized value. If the quantized values are a set of discrete, evenly-spaced grid points, the method is called *uniform quantization*. Otherwise, it is called *non-uniform quantization*. We describe these approaches in detail next.

Approach 1: Symmetric-vs-Asymmetric Uniform Quantization: Uniform quantization maps the embedding table values into integers in the range $[0, 2^n - 1]$. It relies on two parameters: *scale* and *zero_point*. *Scale* specifies the quantization step size, and is defined as $scale = \frac{x_{max} - x_{min}}{2^n - 1}$, while *zero_point* is defined as x_{min}. The quantization proceeds as follows: $x_q = round\left(\frac{x - zero\_point}{scale}\right)$. The de-quantization op-

eration is: $x = scale * x_q + zero\_point$. We denote uniform quantization as $F_Q(x, x_{min}, x_{max})$.

In *symmetric* quantization, x_{max} is set by the maximum absolute value in X, and $x_{min} = -x_{max}$. This is a very simple approach to quantize. An improved approach is to pick x_{min} and x_{max} to use the minimum and maximum element values that are actually present in an embedding vector. We refer to this method as *asymmetric* quantization. Asymmetric quantization, however, has the small additional overhead of storing of both x_{min}, x_{max} values for de-quantization process.

Figure 9 shows the mean ℓ_2 error of symmetric (first bar) and asymmetric quantization (second bar) for different bit-widths used in quantization. Since the elements of the embedding vectors are not symmetrically distributed, asymmetric quantization consistently outperforms symmetric quantization. Note that we generated this result from one representative checkpoint created after training a production dataset for about 18 hours.

Approach 2: Non-uniform Quantization using K-means
We explored non-uniform quantization where embedding vectors are not all mapped into equally spaced buckets. This approach is useful when the elements in a typical embedding vector are not necessarily uniformly distributed.

We leverage the unsupervised K-means clustering algorithm for clustering elements in the embedding vector $X \in \mathbb{R}^n$ into groups. For N-bits *k-means quantization*, the n elements in X are partitioned into 2^N clusters. Let C_i be the cluster i with a corresponding centroid c_i. K-means quantization maps the element $x \in C_i$ to the integer $x_q = i$. In addition, it keeps a codebook entry, such that $codebook[i] = c_i$. The de-quantization operation in that case is: $x = codebook[x_q]$

Figure 9 shows that the third bar in each group, labeled *k-means per vector*, provides lower mean ℓ_2 error compared with asymmetric quantization, when running k-means with 15 iterations. Note that K-means performs slightly worse than asymmetric for a bit-width of 4, due to cluster initialization randomness. While mean ℓ_2 error metric is marginally better, the run time of K-means clustering algorithm was orders of magnitude slower than uniform quantization. For instance, performing K-means clustering using off-the-shelf clustering packages on just one checkpoint of our production training model took more than 48 hours. This is not surprising since prior works have acknowledged the challenge of K-means clustering on large datasets and advocated for sampling a small fraction of the dataset to reduce their overheads [21]. We have explored different approximate clustering strategies but approximations yielded substantial mean ℓ_2 error. Hence, when taking into account any incremental benefits of clustering against the cost of running the clustering algorithm for checkpointing, we conclude that k-means is not feasible in Check-N-Run.

Approach 3: Adaptive Asymmetric Quantization: We observe that the naive way of setting x_{min} and x_{max} in asymmetric quantization may not be optimal in some cases. For example,
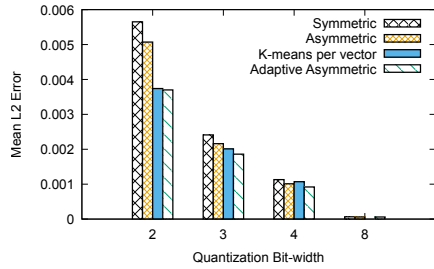
Figure 9: Mean ℓ_2 error of a quantized checkpoint for different quantization approaches
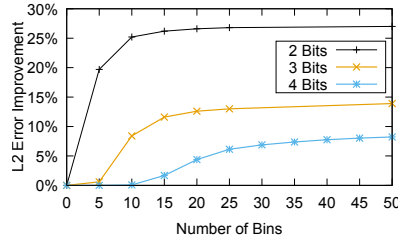
Figure 10: Mean ℓ_2 error improvement of adaptive asymmetric quantization over naive asymmetric quantization for different bit-widths, as a function of bins
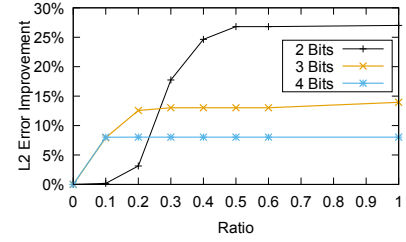
Figure 11: Mean ℓ_2 error improvement for different bit-widths, as a function of the number range ratio (after selecting optimal number of bins)

if a vector contains an element with a relatively high absolute value compared with the other elements, *scale* may be too high.

A brute force approach for selecting more optimal x_{min} and x_{max} values for each embedding vector would iterate over many possible values, and in each iteration perform a quantization for the sole purpose of measuring ℓ_2 error. Based on that, it would choose the x_{min} and x_{max} values that provided the lowest ℓ_2 error. Unfortunately, since this has to be done per embedding vector, it is not feasible in terms of run time when quantizing large models.

To address this issue, Check-N-Run leverages a greedy search algorithm [13] to select the x_{min} and x_{max} values per embedding vector. We define *step_size* as the the original range of the vector divided by a configurable number of bins: $step\_size = \frac{X_{max} - X_{min}}{num\_bins}$. At each iteration, two quantizations are performed for the sole purpose of comparing their ℓ_2 error: $F_Q(x, x_{min} + step\_size, x_{max})$ and $F_Q(x, x_{min}, x_{max} - step\_size)$. Based on the update that provided a lower ℓ_2 error, either x_{min} or x_{max} are set to $x_{min} + step\_size$ or $x_{max} - step\_size$, respectively. Finally, when all iterations are done, the optimal x_{min} and x_{max} are chosen from the iteration with the lowest ℓ_2 error.

The greedy algorithm contains a configurable parameter, *num_bins*, which determines its step size. In addition, we add a *ratio* parameter, which determines the fraction of the original $range = X_{max} - X_{min}$ to iterate over. In other words, the greedy algorithm would iterate as long as $x_{max} - x_{min} < ratio * range$. For example, when *ratio* is set to 1, the algorithm would iterate over the entire range. If *ratio* is 0.6, the algorithm would stop once it covered 60% of the original range. While decreasing the number of bins and ratio both reduce run time, it may also result higher ℓ_2 error. Figure 10 demonstrates the mean ℓ_2 error improvement of adaptive asymmetric quantization over naive asymmetric quantization for different bit-widths, as a function of the number of bins.

Figure 11 depicts the mean ℓ_2 error improvement for various range ratios, based on the optimal number of bins from figure 10 (25 bins for bit-widths of 2 bits and 3 bits, and 45 bins for 4 bits). As can be seen, lower bit-width quantizations are more sensitive to the ratio parameter (and also gain higher improvement by the adaptive asymmetric).

Parameter selection: Check-N-Run automatically sets the greedy algorithm parameters by performing a light-weight checkpoint profiling. It uses the insight that mean ℓ_2 error can be estimated efficiently without having to quantize the entire checkpoint. It uniformly samples a small fraction of the checkpoint (0.001% by default), then quantizes the sampled checkpoint with different parameter values and calculates the mean ℓ_2 errors. With this method, Check-N-Run is able to identify the optimal parameter by automatically choosing the parameter in which the mean ℓ_2 error improvement tapers off. In our experiments, the sampled checkpoint provided identical parameter selection compared with the full checkpoint.

In section 6.1, we evaluate the quantization latency as a function of *num_bins* and *ratio*.

Summary of various approaches: Based on these empirical data, Check-N-Run utilizes adaptive asymmetric quantization for bit-width of 4 bits or less. As shown in figure 9, adaptive asymmetric quantization perform similarly to k-means quantization. For 8-bit quantization, naive asymmetric quantization is sufficient. The quantization bit-width itself is determined dynamically by the expected number of times a training job would resume from a checkpoint, as we elaborate in section 6.

6 Experimental Evaluation

In this section, we evaluate the performance implications of Check-N-Run, its training accuracy implications, and the achieved write bandwidth and storage capacity reduction. We implemented Check-N-Run in our PyTorch training framework and evaluate it in our high-performance training clusters, under production scale models and training datasets.

We use clusters of NVidia HGX-like nodes [25] for training, with some customization such as increased host memory of up to 1.5 TB of DRAM per node, up to 56 cores per node, and alternate scale-out fabric such as NVSwitch and NVLinks (connecting up to 16 nodes). Each GPU is able to communicate directly with GPUs on a different node through a dedicated RoCE NIC, without involving a host CPU. In addition, there is a front-end NIC connected to each CPU. Checkpoints
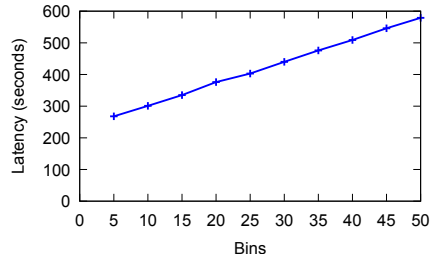
Figure 12: Total checkpoint quantization latency when using adaptive asymmetric quantization, as a function of the number of bins used by the greedy algorithm (ratio=1.0)
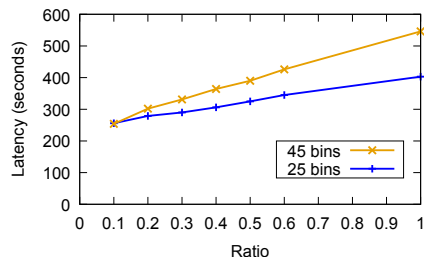

Figure 13: Total checkpoint quantization latency when using adaptive asymmetric quantization, as a function of the ratio used by the greedy algorithm with 25 and 45 bins

are written to remote storage through the regular front-end network, without interfering with inter-GPU communication.

6.1 Performance

Checkpoint overhead on training: Check-N-Run decouples checkpointing from training by creating an in-memory snapshot of the model state before checkpointing. This enables training to continue while checkpoints are created, optimized, and stored in the background. Check-N-Run creates snapshots by copying the model state from GPU's HBM to pinned CPU memory. We measured this operation to take up to 7 seconds in our setting, during which training is stalled. When checkpoint intervals are 30 minutes, the default setting, that overhead translates to less than 0.4% reduction in training throughput.

Tracking the modified embedding vectors in each training iteration requires updating a local bit vector, which is used to mark the modified embedding vectors in the current checkpoint interval. As described in 5.1.1, our efficient implementation uses idle GPU cycles to hide most of this overhead, and reduces the training throughput by less than 1%. Note that these overheads are not dependent on the number of nodes, since nodes typically accommodate roughly the same amount of data, bounded by the GPU's HBM storage capacity (i.e., the number of nodes scales with model size). Hence, larger models do not imply higher snapshot creation or tracking latencies.

Checkpoint quantization latency: Quantization is another source of delay. Since checkpoint quantization is done in

dedicated CPU processes (while training continues in GPUs), it does not affect training throughput. However, it introduces a new latency before the checkpoint can be written to storage. For adaptive asymmetric quantization (used by default for 4 bit and lower quantizations), the overhead is determined by the greedy search parameters. Figure 12 depicts the checkpoint quantization latency of adaptive asymmetric quantization as a function of the number of bins used by the greedy algorithm. The latency to quantize is at most 600 seconds even with 50 bins (the bins are described in section 5.2).

Figure 13 shows the checkpoint quantization latency as a function of the ratio used by the greedy algorithm, using 45 and 25 bins. Increasing the ratio requires searching a wider range of the embedding vector values. As such, the latency grows with ratio.

As a comparison, if we only use asymmetric quantization without the adaptation based on bins and ratio, the latency to quantize is at most 126 seconds. Hence, the "adaptive" approach at least doubles the quantization latency.

Note that the above latency values represent the most pessimistic data. But as explained earlier, quantization in Check-N-Run is performed chunk by chunk (as part of the data serialization, where each chunk contains a small subset of the model state). It is pipelined such that each quantized chunk is written independently to the remote storage, while a new chunk is being quantized. Hence, write bandwidth to remote storage is our main bottleneck, and the observed storage write latency is typically higher than the checkpoint quantization latency. Therefore, the latency of our pipelined quantization approach is virtually zero.

6.2 Accuracy

In this section, we evaluate the training accuracy implications of resuming from a quantized checkpoint using the asymmetric and adaptive asymmetric quantizations described earlier. Since differential checkpointing does not alter training accuracy (all data is preserved on every recovery), we focus this section on quantization approaches only. We use a baseline that does not use quantization to determine accuracy loss of quantization.

Note that the number of stored checkpoints and their frequency do not affect the training accuracy, since training is always done in single-precision floating-point. Quantization is only applied to checkpoints, and would only impact the training job if it resumes from a checkpoint. In that case, Check-N-Run would load a checkpoint and de-quantize it before resuming model training in single precision.

When training jobs have to resume from multiple quantized checkpoints during their lifetime, the quantization error may accumulate. Therefore, the number of times a training job resumes from checkpoints determines the suitable quantization bit-width. Figure 14(a) shows the training lifetime accuracy degradation when loading from a 2-bit quantized
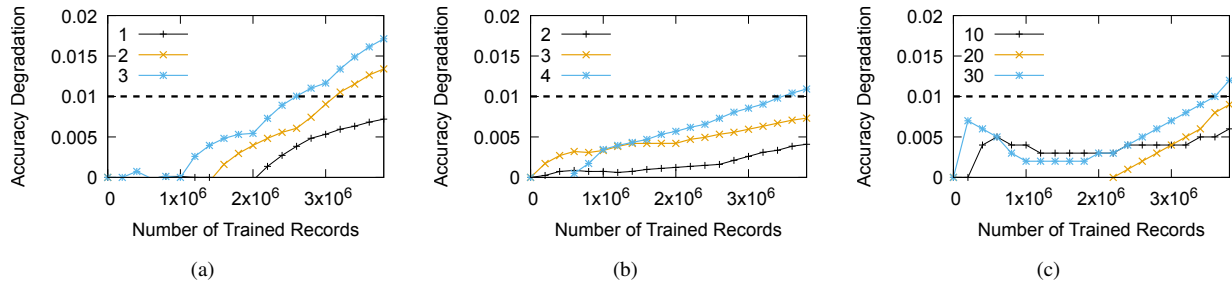
(a)   (b)   (c)

Figure 14: Lifetime accuracy degradation in a training job of 4 billion training samples, when using: (a) 2-bit, (b) 3-bit, and (c) 4bit quantized checkpoints. The lines represent the number of times the job had to resume from a quantized checkpoint
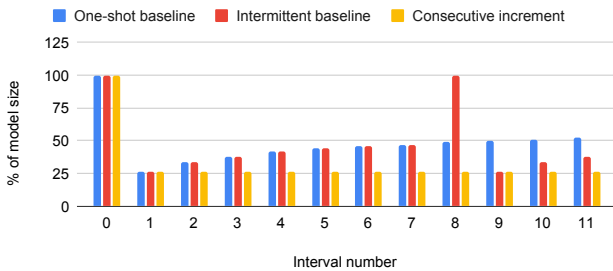

Figure 15: Bandwidth measure: checkpoint size per interval of 30 minutes, for different checkpoint policies
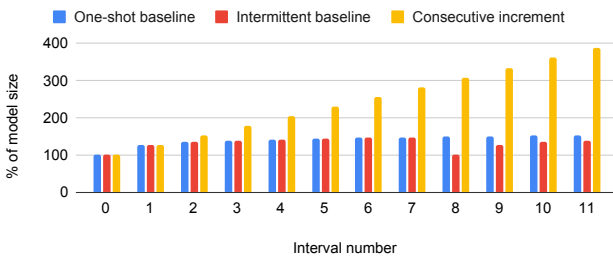

Figure 16: Storage measure: the required storage capacity at each interval of 30 minutes, for different checkpoint policies

checkpoint. We start with 2-bit quantization since it is the most aggressive storage and bandwidth reduction technique of all the approaches. The three lines represent the number of training job failures (failures are uniformly distributed during training), in which the model needs to be reconstructed from a quantized checkpoint. With a single failure, the training accuracy impact is well below the 0.01% threshold even after training with 3 Billion records. However, when two or more failures are encountered during a training run then the 2-bit quantization exceeds the loss threshold of 0.01%.

6.2.1 Dynamic Bit-width Selection:

Figures 14(b) and 14(c) show the accuracy degradation when resuming from 3-bit and 4-bit quantized checkpoints, respectively. As expected, higher bit-widths allow resuming from a checkpoint more times. For 3-bit quantization, a training job may resume from a checkpoint up to 3 times, while for 4-bit quantization one may load the checkpoint up to 20 times. While not shown in the figure, we also measured that with an

8-bit asymmetric quantization, a training job can resume from a checkpoint over 100 times without exceeding the accuracy loss threshold.

Based on the above set of results, Check-N-Run uses a dynamically configurable bit-width selection. Check-N-Run estimates the expected time of training based on the model and the number of nodes. The probability of a node failure in our training cluster (p) is provided as input to Check-N-Run. This probability is computed from failure logs. Check-N-Run then estimates the expected number of failures. Based on this estimate, it picks the bit-width that will not exceed the accuracy threshold. If the number of failures exceeds the estimates during training, Check-N-Run automatically falls back to 8-bit quantization.

6.3 Write Bandwidth and Storage Capacity

In this section, we evaluate the write bandwidth and storage capacity reduction achieved by Check-N-Run, compared with a baseline checkpointing system that uses neither quantization nor differential views.

6.3.1 Differential Checkpointing Policy Comparison

Figure 15 shows the fraction of the model size that is stored in each differential checkpoint, over checkpoint intervals of 30 minutes. This data is a proxy for the bandwidth needed to store the checkpoint. It shows the checkpoint sizes at each interval for different checkpoint policies. In the *One-shot differential* method, the differential checkpoint includes all the embedding vectors that were modified since the first checkpoint, which is created at the first checkpoint interval. As can be seen in the figure, the initial differential checkpoint is only 25% of the total model size, but as the checkpoint size keeps increasing, it exceeds 50% of the model size after 10 intervals. For *Intermittent differential* method, the figure shows how the checkpoint size increases until Check-N-Run dynamically switches to taking a full baseline checkpoint at interval 8, just before the checkpoint size reaches 50% of the model size. The new baseline checkpoint includes the entire model, but the next checkpoint size is only about 25% of the full model size
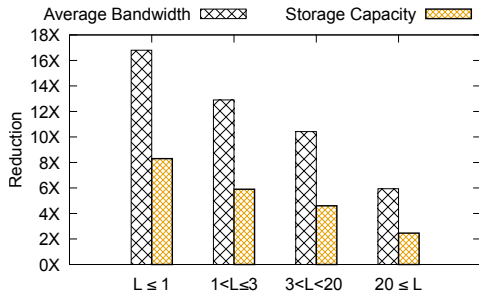
Figure 17: Overall reduction of the checkpoint average write bandwidth and storage capacity. *L* represents the number of times the training job had to resume from a checkpoint.

Figure 16 shows the total required storage capacity (relative to the model size), over several checkpoint intervals of 30 minutes. The *One-shot differential* approach includes the first checkpoint taken and the latest differential checkpoint at each interval. As expected, the consumed capacity increases over time. The reason is that every differential checkpoint stores *all* the modified entries since the first checkpoint, along side the first checkpoint itself. In the case of *Intermittent differential*, the required capacity increases until the full checkpoint is triggered at interval 8. At that point, the consumed storage capacity resets and includes only the newly taken full checkpoint.

Figures 15 and 16 also show the impact of the *Consecutive incremental* policy, which only stores the vectors that were modified in the current checkpoint interval. The recovery process is more complex, since all previous checkpoints must be read for recovery. As can be seen, this approach reduces the size of checkpoints over time and the corresponding write bandwidth (e.g., the average write bandwidth in a duration of 12 intervals is 33% less than the other policies). Moreover, the checkpoint size is stable, since the number of vectors that are updated during an interval stays roughly the same. However, since all the checkpoints have to be kept, the required storage capacity increases rapidly, reaching almost ×4 the model size after only 11 intervals. As such, Check-N-Run uses the intermittent differential policy by default.

6.3.2 Overall Reduction

Figure 17 presents the overall reduction in write bandwidth and storage capacity, when combining both quantization and differential checkpointing (intermittent baseline policy), and using the thresholds from section 6.2.1 for selecting the quantization bit-width. When a training job is expected to resume from checkpoint no more than one time, Check-N-Run reduces the average consumed write bandwidth and maximum storage capacity by 17× and 8×, respectively. Even in the not so common case of more than 20 failures, Check-N-Run reduces the average bandwidth by 6× and the maximum storage capacity by 2.5×. Note that these savings are not linearly proportional to the chosen quantization bit-width due to

the metadata structure. That structure includes the differential checkpoint index and quantization parameters. Metadata structure can be further optimized in future work.

7 Related Work

Checkpointing has been explored in many distributed systems [2, 17, 27, 28, 35]. Checkpoint optimization schemes include techniques to reduce latency [31], coordinating across multiple snapshots for efficient reconstruction [27, 35], using different checkpoint resolutions for providing varying levels of recovery [8, 20]. The goal of Check-N-Run is to deal with checkpoints that are terabytes in size. As such, reducing storage and network bandwidth is important. Unlike traditional distributed systems, where getting a consistent view across different machines is a challenge [2, 28], Check-N-Run exploits the repetitive nature of synchronous training to initiate checkpoints at the end of a training batch.

In terms of ML-specific checkpointing, Deepfreeze [24] checkpoints DNN models using variable resolution, while handling storage-specific API and sharding needs. Microsoft's ADAM uses zip compression to reduce checkpoint size of DNN models [5]. CheckFreq uses dynamic rate tuning to automatically decide when to initiate a checkpoint and a decoupled store-train pipleine [19]. Check-N-Run tackles reducing storage and bandwidth needs through quantization combined with incremental view. Similar to CheckFreq, it also decouples checkpoint processing from training.

Quantization has been applied to ML models, particularly in the context of inference. Prior works used floating to fixed point quantization to improve compute efficiency [18], ternary quantization for inference on mobile devices [37, 40], per-layer heterogeneous quantization of DNNs [39], mixed precision quantization that adapts to underlying hardware capabilities [34], quantization of gradient vectors for bandwidth efficient aggregation [1, 9, 36], lossy training using 1-bit quantization [29] and more. To the best of our knowledge, using quantization to reduce checkpoint size of recommendation models has not been made public.

8 Conclusion

This paper presents Check-N-Run, a high-performance checkpointing system for training recommendation systems at scale. The primary goal of Check-N-Run is to reduce the bandwidth and storage costs without compromising accuracy. Hence, Check-N-Run leverages differential checkpointing and dynamically selected quantization techniques to significantly reduce the required write bandwidth and storage capacity for checkpointing real-world models. Our evaluations show that depending on the number of recovery events one may need to adapt quantization of different bit widths. By combining such adaptive quantization with differential checkpointing, Check-N-Run provides 6-17x reduction in required bandwidth, while simultaneously reducing the storage capacity by 2.5-8X.

References

[1] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. Qsgd: Communication-efficient sgd via gradient quantization and encoding. In *Advances in Neural Information Processing Systems*, pages 1709–1720, 2017.

[2] K Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985.

[3] Jianmin Chen, Rajat Monga, Samy Bengio, and Rafal Józefowicz. Revisiting distributed synchronous SGD. *CoRR*, abs/1604.00981, 2016.

[4] Suming J. Chen, Zhen Qin, Zac Wilson, Brian Calaci, Michael Rose, Ryan Evans, Sean Abraham, Donald Metzler, Sandeep Tata, and Mike Colagrosso. Improving recommendation quality in google drive. In *KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020*, pages 2900–2908. ACM, 2020.

[5] Trishul M. Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, pages 571–582. USENIX Association, 2014.

[6] Yann Collet and Chip Turner. Smaller and faster data compression with zstandard. *http://www. rgoarchitects. com/Files/fallacies. pdf*, 2016.

[7] Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems, Boston, MA, USA, September 15-19, 2016*, pages 191–198. ACM, 2016.

[8] Sheng Di, Mohamed-Slim Bouguerra, Leonardo Arturo Bautista-Gomez, and Franck Cappello. Optimization of multi-level checkpoint model for large scale HPC applications. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*, pages 1181–1190. IEEE Computer Society, 2014.

[9] Nikoli Dryden, Tim Moon, Sam Ade Jacobs, and Brian Van Essen. Communication quantization for data-parallel training of deep neural networks. In *2016 2nd Workshop on Machine Learning in HPC Environments (MLHPC)*, pages 1–8. IEEE, 2016.

[10] Assaf Eisenman, Maxim Naumov, Darryl Gardner, Misha Smelyanskiy, Sergey Pupyrev, Kim M. Hazelwood, Asaf Cidon, and Sachin Katti. Bandana: Using non-volatile memory for storing deep learning models. In *Proceedings of Machine Learning and Systems 2019, MLSys 2019, Stanford, CA, USA, March 31 - April 2, 2019*. mlsys.org, 2019.

[11] Andrey Goder, Alexey Spiridonov, and Yin Wang. Bistro: Scheduling data-parallel jobs against live production systems. In *2015 USENIX Annual Technical Conference, USENIX ATC '15, July 8-10, Santa Clara, CA, USA*, pages 459–471. USENIX Association, 2015.

[12] Carlos Alberto Gomez-Uribe and Neil Hunt. The netflix recommender system: Algorithms, business value, and innovation. *ACM Trans. Manag. Inf. Syst.*, 6(4):13:1–13:19, 2016.

[13] Hui Guan, Andrey Malevich, Jiyan Yang, Jongsoo Park, and Hector Yuen. Post-training 4-bit quantization on embedding tables. In *MLSys Workshop on Systems for ML @ NeurIPS*, 2019.

[14] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim Hazelwood, Mark Hempstead, Bill Jia, et al. The architectural implications of facebook's dnn-based personalized recommendation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 488–501. IEEE, 2020.

[15] Robert L Henderson. Job scheduling under the portable batch system. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 279–294. Springer, 1995.

[16] Samuel Hsia, Udit Gupta, Mark Wilkening, Carole-Jean Wu, Gu-Yeon Wei, and David Brooks. Cross-stack workload characterization of deep recommendation systems. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*, pages 157–168. IEEE, 2020.

[17] R Koo and S Toueg. Checkpointing and recovery rollback for distributed systems. *IEEE Transactions on Software Engineering*, 13(1):23–31, 1987.

[18] Darryl Lin, Sachin Talathi, and Sreekanth Annapureddy. Fixed point quantization of deep convolutional networks. In *International conference on machine learning*, pages 2849–2858. PMLR, 2016.

[19] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. Checkfreq: Frequent, fine-grained DNN checkpointing. In Marcos K. Aguilera and Gala Yadgar, editors, *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*, pages 203–216. USENIX Association, 2021.

[20] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R De Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2010.

[21] Laurence Morissette and Sylvain Chartier. The k-means clustering technique: General considerations and implementation in mathematica. *Tutorials in Quantitative Methods for Psychology*, 9(1):15–24, 2013.

[22] Maxim Naumov, John Kim, Dheevatsa Mudigere, Srinivas Sridharan, Xiaodong Wang, Whitney Zhao, Serhat Yilmaz, Changkyu Kim, Hector Yuen, Mustafa Ozdal, et al. Deep learning training in facebook data centers: Design of scale-up and scale-out systems. *arXiv preprint arXiv:2003.09518*, 2020.

[23] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*, 2019.

[24] Bogdan Nicolae, Jiali Li, Justin Wozniak, George Bosilca, Matthieu Dorier, and Franck Cappello. Deepfreeze: Towards scalable asynchronous checkpointing of deep learning models. In *CCGrid'20: 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing*, 2020.

[25] Nvidia. Nvidia hgx2 datasheet. *https://images.nvidia.com/content/pdf/hgx2-datasheet.pdf*.

[26] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2009.

[27] Fabrizio Petrini, Kei Davis, and José Carlos Sancho. System-level fault-tolerance in large-scale parallel machines with buffered coscheduling. In *18th International Parallel and Distributed Processing Symposium (IPDPS 2004), CD-ROM / Abstracts Proceedings, 26-30 April 2004, Santa Fe, New Mexico, USA*. IEEE Computer Society, 2004.

[28] James S Plank. An overview of checkpointing in uniprocessor and distributed systems, focusing on implementation and performance. Technical report, UT-CS-97-372, 1997.

[29] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns.

In *INTERSPEECH 2014, 15th Annual Conference of the International Speech Communication Association, Singapore, September 14-18, 2014*, pages 1058–1062. ISCA, 2014.

[30] Brent Smith and Greg Linden. Two decades of recommender systems at amazon.com. *IEEE Internet Comput.*, 21(3):12–18, 2017.

[31] Nitin H Vaidya. *On checkpoint latency*. Citeseer, 1995.

[32] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

[33] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. Billion-scale commodity embedding for e-commerce recommendation in alibaba. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018*, pages 839–848. ACM, 2018.

[34] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. Haq: Hardware-aware automated quantization with mixed precision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8612–8620, 2019.

[35] Long Wang, Karthik Pattabiraman, Zbigniew Kalbarczyk, Ravishankar K Iyer, Lawrence Votta, Christopher Vick, and Alan Wood. Modeling coordinated checkpointing for large-scale supercomputers. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 812–821. IEEE, 2005.

[36] Mingchao Yu, Zhifeng Lin, Krishna Narra, Songze Li, Youjie Li, Nam Sung Kim, Alexander G. Schwing, Murali Annavaram, and Salman Avestimehr. Gradiveq: Vector quantization for bandwidth-efficient gradient aggregation in distributed CNN training. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 5129–5139, 2018.

[37] Dongqing Zhang, Jiaolong Yang, Dongqiangzi Ye, and Gang Hua. Lq-nets: Learned quantization for highly accurate and compact deep neural networks. In *Proceedings of the European conference on computer vision (ECCV)*, pages 365–382, 2018.

[38] Weijie Zhao, Jingyuan Zhang, Deping Xie, Yulei Qian, Ronglai Jia, and Ping Li. Aibox: Ctr prediction model training on a single node. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, pages 319–328, 2019.

[39] Yiren Zhou, Seyed-Mohsen Moosavi-Dezfooli, Ngai-Man Cheung, and Pascal Frossard. Adaptive quantization for deep neural network. *arXiv preprint arXiv:1712.01048*, 2017.

[40] Chenzhuo Zhu, Song Han, Huizi Mao, and William J Dally. Trained ternary quantization. *arXiv preprint arXiv:1612.01064*, 2016.

MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters

Qizhen Weng[†*], Wencong Xiao[*], Yinghao Yu[*†], Wei Wang[†], Cheng Wang[*],
Jian He[*], Yong Li[*], Liping Zhang[*], Wei Lin[*], and Yu Ding[*]

[†]Hong Kong University of Science and Technology   [*]Alibaba Group

{qwengaa, weiwa}@cse.ust.hk, {wencong.xwc, yinghao.yyh, wc189854, jian.h, jiufeng.ly, liping.z, weilin.lw, shutong.dy}@alibaba-inc.com

Abstract

With the sustained technological advances in machine learning (ML) and the availability of massive datasets recently, tech companies are deploying large ML-as-a-Service (MLaaS) clouds, often with heterogeneous GPUs, to provision a host of ML applications. However, running diverse ML workloads in heterogeneous GPU clusters raises a number of challenges. In this paper, we present a characterization study of a two-month workload trace collected from a production MLaaS cluster with over 6,000 GPUs in Alibaba. We explain the challenges posed to cluster scheduling, including the low GPU utilization, the long queueing delays, the presence of hard-to-schedule tasks demanding high-end GPUs with picky scheduling requirements, the imbalance load across heterogeneous machines, and the potential bottleneck on CPUs. We describe our current solutions and call for further investigations into the challenges that remain open to address. We have released the trace for public access, which is the most comprehensive in terms of the workloads and cluster scale.

1 Introduction

Driven by recent algorithmic innovations and the availability of massive datasets, machine learning (ML) has achieved remarkable performance breakthroughs in a multitude of real applications such as language processing [23], image classification [33,55], speech recognition [32,56,62], and recommendation [30,60,74]. Today's production clusters funnel large volumes of data through ML pipelines. To accelerate ML workloads at scale, tech companies are building fast parallel computing infrastructures with a large fleet of GPU devices, often shared by multiple users for improved utilization and reduced costs. These large GPU clusters run all kinds of ML workloads (e.g., training and inference), providing infrastructure support for ML-as-a-Service (MLaaS) cloud [2–4,7,8].

In this paper, we share our experiences in running ML workloads in large GPU clusters. We present an extensive

characterization of a two-month workload trace[1] collected from a production cluster with 6,742 GPUs in Alibaba PAI (Platform for Artificial Intelligence) [2]. The workloads are a mix of training and inference jobs submitted by over 1,300 users, covering a wide variety of ML algorithms including convolutional and recurrent neural networks (RNNs and CNNs), transformer-based language models [23,37,56], GNNs-based (graph neural network) recommendation models [31,57,75], and reinforcement learning [39,43,44]. These jobs run in multiple ML frameworks, have different scheduling requirements like GPU locality and gang scheduling, and demand variable resources in a large range spanning orders of magnitude. GPU machines are also heterogeneous (see Table 1) in terms of hardware (e.g., V100, P100, T4) and resource configurations (e.g., GPUs, CPUs, and memory size). In comparison, prior workload analyses focus mainly on training CNN and RNN models in homogeneous environments [18,29,36,41,65,66,72].

The large heterogeneity of ML workloads and GPU machines raises a number of challenges in resource management and scheduling, making it difficult to achieve high utilization and fast job completion. We present those challenges, describe our solutions to some of them, and invite further research on the open problems.

Low utilization caused by fractional GPU uses. In our cluster, a task instance usually can only use parts of a GPU. In fact, the median usage of streaming multiprocessors (SMs) of an instance is 0.042 GPUs. Existing coarse-grained GPU allocation schemes dedicate an entire GPU to one task instance [36,41,72], and would result in extremely low utilization in our cluster.

We address this problem with *GPU sharing*, a technique that allows multiple ML tasks to time-multiplex a GPU in a controlled manner [66]. Utilizing this feature, the scheduler consolidates a large volume of low-GPU workloads onto a small number of machines, using only 50% of the requested

---

[1]The trace was collected in July and August 2020, and is now open for public access as part of the Alibaba Cluster Trace Program [1].

GPUs on average. Such consolidation causes no severe interference: among high-utilization GPUs, only 4.5% run ML tasks with potential contention on SMs.

Long queueing delays for short-running task instances. Short-running task instances are prone to long queueing delays caused by head-of-line blocking. In fact, around 9% of short-lived instances spent more than half of their completion time waiting to be scheduled. An effective solution is to predict the task run-time and prioritize short tasks over the long ones. Existing approaches require specialized framework support to track and estimate the training progress [41, 46, 49], which is not always possible in production as users can run standard or customized ML frameworks without such feature.

However, there is a silver lining. In our cluster, the majority of workloads are recurring, with 65% of tasks repeatedly executed at least 5 times in the trace. Through careful feature engineering, we can predict the durations of most recurring tasks within 25% error, sufficient to make quality scheduling decisions as suggested by previous work [16]. Trace-driven simulations shows that using shortest-job-first scheduling with predicted task durations reduces the average completion time by over 63%.

Hard to schedule high-GPU tasks. Our cluster runs a small portion of compute-intensive ML tasks for business-critical, user-facing applications. These tasks request full GPUs (no sharing) and can attain dramatic speedup on high-end devices by exploiting advanced hardware features such as NVLink [12] (see Section 6.1)—these picky requirements make them difficult to schedule.

Our scheduler employs a simple *reserving-and-packing* policy to differentiate those hard-to-schedule high-GPU tasks from other tasks. It reserves high-end GPU machines (e.g., V100 with NVLinks) for a small number of high-GPU tasks with picky scheduling requirements, while packing the other workloads on less advanced machines, using GPU sharing. The reserving-and-packing policy reduces the average queueing delay by 68% for high-GPU tasks and 45% for all.

In our quest for optimized cluster management, a few challenges remain open, which have received less attention in the literature.

Load imbalance. We observe imbalanced load running in heterogeneous machines. In general, machines with low-end GPUs are more crowded than those with high-end GPUs: the former have over 70% CPUs and GPUs of these machines allocated on average, while the latter have only 35% CPUs and 49% GPUs allocated. There is also a *provisioning mismatch* between workloads and machines. On average, workloads running in 8-GPU machines demand $1.9\times$ more CPUs per GPU than the machines can provide (12 CPUs per GPU), whereas those running in 2-GPU machines request 53% fewer CPUs per GPU than the machine specifications (32 or 48 CPUs per GPU).

Bottleneck on CPUs. While ML workloads perform training and inference on GPUs, many data processing (e.g., data fetching, feature extraction, sampling) and simulation tasks (e.g., reinforcement learning) involved in the pipeline run on CPUs, which can also become a bottleneck. In fact, we find that workloads running in machines with higher CPU utilization are more likely to get slowdown. For example, in T4 machines, those slowed tasks measure an average of 33.5% P75 CPU utilization, noticeably higher than that measured by the accelerated tasks (21.3%). Similar results are also found in V100 machines reserved for high-GPU workloads (50.6% P75 CPU utilization for slowed tasks and 42.4% for the accelerated), indicating that even GPU-demanding workloads can be harmed by CPU contention.

We believe the observations made in our cluster do not stand in isolation. We share the insights derived from our analysis and discuss potential system optimization opportunities in improving ML framework, adopting resource disaggregation, and decoupling data pre-processing from GPU training (see Section 7). We hope that the observations and experiences shared in our study, as well as the release of the PAI trace, can inspire follow-up research in optimizing ML workload scheduling and GPU cluster management.

2 Background

Fast growing data and GPU demand. The support for scalable machine learning has become increasingly important in production data processing pipelines. In our experience of operating general-purpose ML platforms for production workloads, we have witnessed the fast growing demand of both training data and GPU resources. In just a few years, the sheer volume of training data for an ML job has grown orders of magnitude, from the standard dataset of 100s GB (e.g., ImageNet [22]) to an Internet scale of 10s or even 100s TB. The massive volume of data forces ML jobs to scale out to a large number of GPU machines. In our cluster, the largest single ML job requests to run on over 1,000 GPUs, posing a significant gang-scheduling challenge to the cluster.

Alibaba PAI. To accommodate the fast growing computing demand of ML workloads, Alibaba Cloud offers Machine Learning Platform for AI (PAI), an all-in-one MLaaS platform that enables developers to use ML technologies in an efficient, flexible, and simplified way. PAI provides various services covering the entire ML pipeline, including feature engineering, model training, evaluation, inference, and autoML. Since its introduction in 2018, PAI has gained tens of thousands of enterprises and individual developers, making it one of the largest leading MLaaS platforms in China.

Figure 1 illustrates an architecture overview of PAI, where users submit ML jobs developed in a variety of frameworks, such as TensorFlow [14], PyTorch [48], Graph-Learn [75], RLlib [38]. Upon the job submission, users provide the application code and specify the required compute resources, such
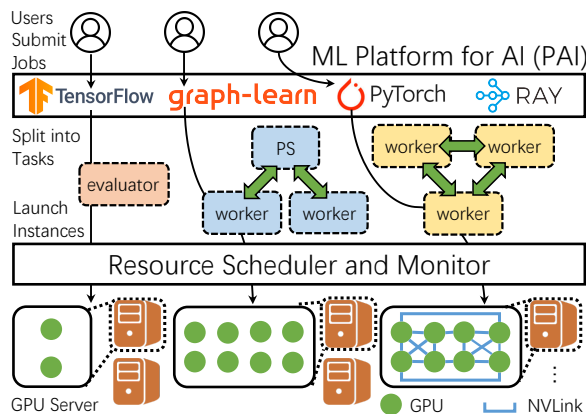
Figure 1: Architecture overview of PAI.

Table 1: Machine specs of GPU clusters in the existing trace analysis works. GPUs with † are equipped with NVLink [12]. The Philly trace does not reveal CPU specs and GPU types.

System	#CPUs	Mem (GiB)	#GPUs	GPU type	#Nodes
PAI	64	512	2	P100	798
	96	512	2	T4	497
	96	512	8	Misc.	280
	96	384	8	V100M32†	135
	96	512/384	8	V100†	104
	96	512	0	N/A	83
Philly [36]	Unk.	528/264	2	12GB GPU	321
	Unk.	528/264/132	8	24GB GPU	231
Tiresias [29]	20	256	4	P100†	15
Gandiva$_{fair}$ [18]	12	224	4	K80	32
	12	448	4	P100	12
	12	448	4	V100	6
Themis [41]	24	448	4	K80	12
	12	224	2	K80	8
HiveD [72]	24	224	4	K80	125
	24	224	4	M60	75
Antman [66]	96	736	8	V100M32†	8

as GPUs, CPUs, and memory. Each job is translated into multiple *tasks* of different roles, such as parameter servers (PS) and workers for a training job, and evaluator for an inference job. Each task may consist of one or multiple instances and can run on multiple machines. PAI employs Docker containers to instantiate tasks for simplified scheduling and execution on heterogeneous hardware.

Trace analysis. Running diverse ML workloads in shared GPU clusters at cloud scale raises daunting challenges. Trace analysis is essential to understand those challenges and provide new insights on system optimization. However, existing analyses are performed on GPU clusters with limited size, workload diversity, and machine heterogeneity, and hence cannot fully represent the state of the art (see Table 1). Take Microsoft's Philly trace [36] as an example. Whereas distributed training is now commonplace, the majority of Philly
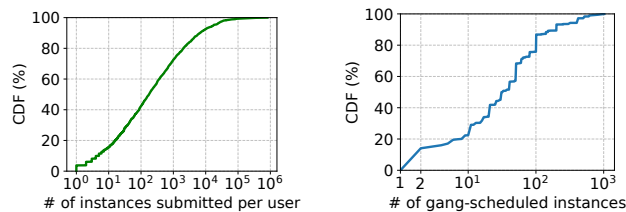
(a) CDF of the number of instances run by a user.

(b) CDF of gang-scheduled task instances.

Figure 2: Heavily skewed distribution of task instances run by users and the prevalence of gang-scheduling requirements.

workloads ($> 82\%$) ran on a single GPU instance when the trace was collected in 2017. It is also unclear what types of GPUs were used to run those workloads, which may have significant impact to scheduling [41, 46]: the performance of new-generation GPUs can be 1.1–$8\times$ higher than the older generations [18]. Moreover, the Philly trace only includes the training workloads, whereas it is common to run both training and inference jobs in a shared MLaaS platform [47, 51, 69].

The insufficiency of existing works motivates the release of the PAI trace, which we examine next.

3 Workload Characterization

In this section, we analyze the ML workloads in the released PAI trace. We start with an overview of the trace, followed by a characterization of its temporal and spatial patterns.

3.1 Trace Overview

Trace information. The released PAI trace contains a hybrid of training and inference jobs running state-of-the-art ML algorithms in mainstream frameworks [14, 48, 75]. Most jobs request multiple GPUs. The trace records the arrival time, completion time, resource requests and usages in GPUs, CPUs, GPU memory and main memory of the workloads at various levels (e.g., job, task, and instance) (Sections 3.2 and 3.3). The application semantics, such as whether the code is performing training or inference, and in what ML framework, are not available as our cluster scheduling system Fuxi [26, 71] only sees the execution containers and is agnostic to the running applications. Nevertheless, we have manually examined some workloads and included their application names (e.g., click-through rate prediction and reinforcement learning) in the trace to provide some clues whenever possible (Sections 6.1 and 6.2). Machine-level information is also provided in the trace, including the hardware specs (Table 1) and time-varying resource utilizations (Section 4) collected by the daemon agents that periodically query the Linux kernel and GPU driver (e.g., NVIDIA Management Library [9]) in the host machines. The detailed schema and trace data are given in the trace repository [1].
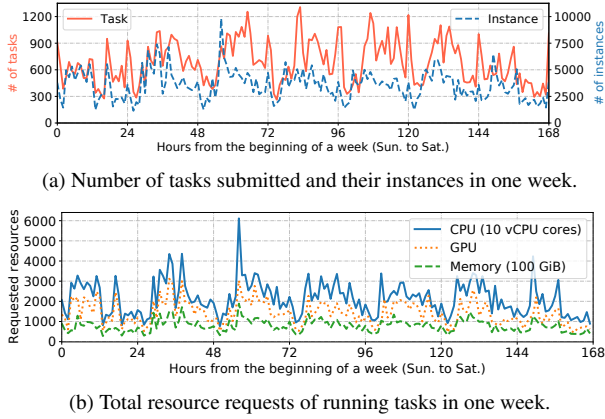
(a) Number of tasks submitted and their instances in one week.

(b) Total resource requests of running tasks in one week.

Figure 3: Task submissions and resource requests roughly follow diurnal patterns.

Jobs, tasks, and instances. In PAI, users submit *jobs*. Each job has one or multiple *tasks* taking different computation roles. Each task runs one or multiple *instances* in Docker containers. For example, a distributed training job may have a parameter-server (PS) task of 2 instances and a worker task of 10 instances. All instances of a task have the same resource demands and might be *gang-scheduled* (e.g., running simultaneously for all PyTorch workers). Our characterization in this subsection mainly focuses on task instances.

Heavy-skewed instance distribution. The PAI trace contains more than 7.5 million instances of 1.2 million tasks submitted by over 1,300 users. Figure 2a depicts the distribution of task instances run by users, which is *heavily skewed*. More specifically, around 77% of task instances are submitted by the top 5% users, each running over 17.5k instances, while the bottom 50% users run less than 180 instances each.

The prevalence of gang-scheduling. Our distributed ML jobs require gang-scheduling. As shown in Figure 2b, among all task instances, around 85% have such requirements, in which 20% must be gang-scheduled on more than 100 GPUs, some even requesting over 1,000. Together, tasks with gang-scheduled instances account for 79% of the total GPU demands. The prevalence of these tasks makes it difficult to achieve high utilization.

GPU locality. In addition to gang-scheduling, a task may request to run all its instances on multiple GPUs co-located in one machine, a requirement known as *GPU locality*. Although such requirement often leads to prolonged scheduling delays [29, 36, 72], it enables the use of high-speed GPU-to-GPU interconnect within a single node (e.g., NVLink and NVSwitch), which can dramatically accelerate distributed training [12, 15, 36]. In our cluster, enforcing GPU locality yields over 10× speedup for some training tasks (Section 6.1).

GPU sharing. PAI supports *GPU sharing* that allows multiple task instances to time-share a GPU at a low cost. With this feature, users can specify GPU request in $(0, 1)$ and run

(a) CDF of instance run-time.

(b) CDF of normalized instance queueing delays.

(c) CDF of queueing delays w.r.t. GPU requests per instance.

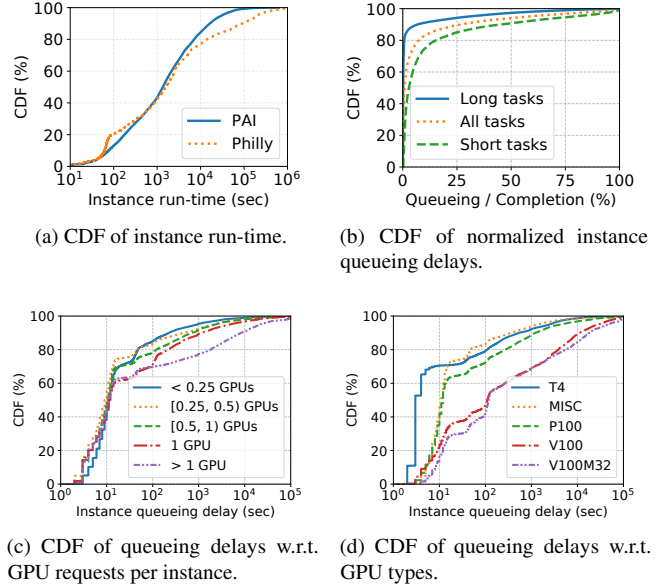(d) CDF of queueing delays w.r.t. GPU types.

Figure 4: CDF of instance run-time and queueing delays.

its task instances using parts of GPUs. We will show in Section 5.1 that GPU sharing enables considerable savings on GPU provisioning.

Various GPU types to choose from. PAI provides heterogeneous GPUs and allows users to specify the required GPU types to run their tasks. The available choices include NVIDIA Tesla T4, P100, V100, V100M32 (V100 SXM2 with 32 GB memory), and other GPUs of older generations (Misc in Table 1), e.g., Tesla K40m, K80, and M60. In our cluster, only 6% tasks require to run on specified GPUs, while the others have no such limitation and can run on any GPUs.

3.2 Temporal Pattern

We next examine the temporal patterns of the PAI workloads.

Diurnal task submissions and resource requests. Figure 3 depicts task and instance submissions as well as the overall resource requests in one week during the trace collection period. We observe rough diurnal patterns, where task submissions in weekdays (from the 24th to 144th hours) are slightly higher than in weekends. It is worth mentioning that in addition to the daytime, midnight is also a rush hour for task submissions (Figure 3a). Yet, most tasks submitted at midnight are less compute-intensive, having only a few instances and requesting a small amount of resources (Figure 3b).

Instance run-time in a wide range. Figure 4a shows the distribution of instance run-time (solid line). Similar to the Philly trace [36] (dotted line), instance run-time varies in a wide range spreading four orders of magnitude. The median run-time (23 minutes) is comparable with that of Philly (26 minutes), while their 90th percentile (P90) run-time (4.5 hours) is shorter than that of Philly (25 hours).

Non-uniform queueing delays. The queueing delay (aka wait time or scheduling delay), measured from the moment of task submission to the start of the task instance, varies greatly among instances. Compared to the long-running instances, short-running instances usually spend a larger portion of time in queueing. To see this, we use the median run-time as a threshold and divide instances into long-running and short-running ones, where a long-running (short-running) task instance has a longer (shorter) run-time than the median. In Figure 4b, We compare the queueing delays of these task instances relative to their completion times (queueing delay plus run-time). Around 9% short-running instances spend more than half of the completion time waiting to be scheduled; this number drops to 3% when it comes to long-running instances.

A task instance's queueing delay also depends on its GPU request. Figure 4c shows that instances willing to share GPUs (i.e., GPU request in $(0,1)$) can be quickly scheduled, with the 90th percentile (P90) queueing delay being 497 seconds. In comparison, instances that do not accept GPU sharing need to wait for a longer time, with the P90 delay being 1,150 (8,286) seconds for those requesting one GPU (> 1 GPU).

Long queueing delays are also seen in instances requesting high-end GPUs. As shown in Figure 4d, for instances running on advanced V100 GPUs (including V100M32), the median and P90 delays are 113 and 13,709 seconds, respectively. In comparison, for instances running on low-end miscellaneous GPUs, the median and P90 delays are only 11 and 360 seconds, respectively.

3.3 Spatial Pattern

We finally present the spatial patterns of the PAI task instances by analyzing their resource requests and usages. PAI collects the system metrics of running tasks every 15 seconds and provides visualization tools [2, 25] for users to analyze the workload patterns and figure out their resource requests.

Heavy-tailed distribution of resource requests. Figures 5a, 5b, and 5c (blue solid lines) respectively depict the distributions of the total CPUs, GPUs, and memory requested by all instances. All three distributions are heavy-tailed, with around 20% instances requesting large resource amounts and the other 80% requesting small to medium. More specifically, the P95 request demands 12 vCPU cores[2], 1 GPU, and 59 GiB memory, more than twice the median request (6 vCPU cores, 0.5 GPUs, and 29 GiB memory).

Uneven resource usage: Low on GPU but high on CPU. Most users tend to ask for more resources than they actually need, resulting in a low resource usage (dotted lines in Figures 5a, 5b, and 5c). In our cluster, the median instance resource usages are 1.4 vCPU cores, 0.042 GPUs, and 3.5 GiB memory, much smaller than the median request. We stress

---

[2]In our cluster, each physical processor core consists of two vCPU cores, using hyper-threading technology [42].

(a) CPU request and usage. (b) GPU request and usage.

(c) Memory request and usage. (d) Usage minus request, normalized by the machine capacity.
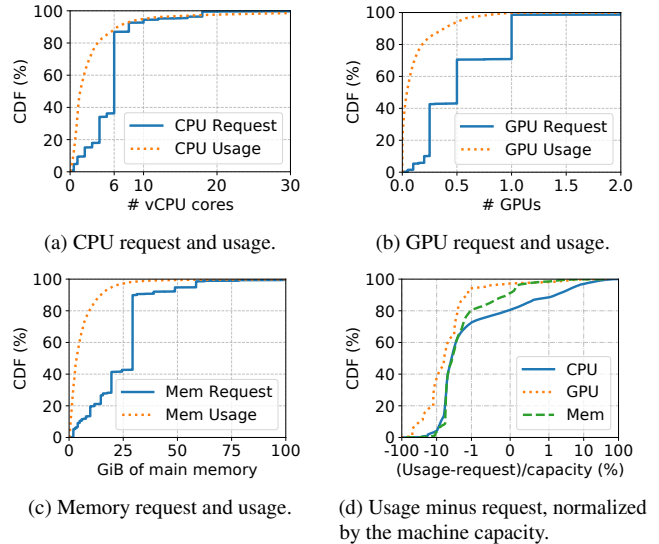
Figure 5: CDF of instance resource requests and actual usages.

that the low GPU usage is not caused by the low computing demand, but by contentions on other resource like CPU, making GPUs idle for most of the time (Section 6.2). Figure 5b also shows that around 18% instances barely use GPUs: they perform computations such as running parameter servers, fetching and pre-processing data, which are mostly on CPUs with small or no GPU involvement.

In PAI, instances of a task can use *spare resources* in the host machines, making it possible to *overuse* more resources than requested. Compared to GPU and memory, overuse of CPUs is more prevalent. To see this, for each instance we measure the difference between its resource usage and request for CPU, GPU, and memory—positive (negative) being overuse (underuse). We normalize the results by the machine's CPU, GPU, and memory capacity, respectively, and depict the distributions in Figure 5d. There are 19% task instances overusing CPUs (blue solid line with $X > 0$). In comparison, only 3% (9%) instances use more GPUs (memory) than they requested.

4 GPU Machine Utilization

Having studied the workload characterization, we turn to resource utilization in GPU machines.

4.1 Utilization of Compute Resources

We start to analyze the utilization of compute resources, including CPU, GPU, main and GPU memory. Our cluster has 1,295 2-GPU machines and 519 8-GPU machines (Table 1). Machines with 8 GPUs have a lower CPU-to-GPU ratio than those with 2 GPUs. In light of their different configurations, we perform measurement separately for the two types of machines. Each machine has time series data of resource utilization measured every 15 seconds by the monitoring system.
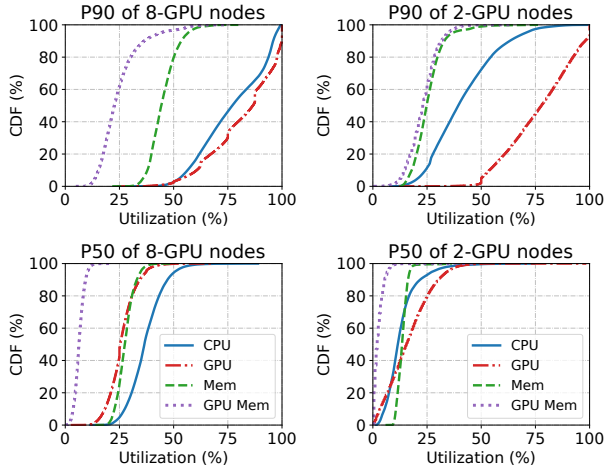
Figure 6: CDF of P90 and P50 (median) utilization of CPU, GPU, main and GPU memory in different machine groups.

At each timestamp, we collect the utilization of all 8-GPU machines and calculate the tail (P90) and the median (P50). Together, we obtain a sequence of P90 and P50 utilizations taken at different timestamps. We depict their distributions in Figure 6 (two subfigures on the left). We perform the same measurements in 2-GPU machines and depict the results in the right two subfigures. Compared to memory (main and the GPU's), GPU and CPU have higher utilization. In 8-GPU machines (upper-left in Figure 6), the average P90 utilization of GPU (red dash-dotted line) and CPU (blue solid line), i.e., the arithmetic mean of P90 values from all timestamps, reaches 82% and 77%, respectively. In 2-GPU machines (upper-right in Figure 6), the P90 GPU utilization remains high (77% on average), while the P90 CPU utilization drops to 42% on average due to the large CPU-to-GPU ratio (32 or 48 CPUs per GPU). In both types of machines, the P90 utilization of the main and GPU memory stays below 60% at almost all time, indicating that our tasks are less memory-intensive.

Compared to other resources, we measure a larger variation of utilization on GPUs. As shown in Figure 6, the distribution of P90 GPU utilization spans a wide range from less than 40% to 100% of the computing power provided by the streaming multiprocessors of the machine's GPUs; the difference between the tail and the median utilization is also larger on GPU than on other resources (comparing the top sub-figures with the bottom). The large variation is partly due to the bursty GPU usage patterns found in our ML workloads [65, 66]. It is also due to the design of our scheduler that prioritizes packing over load balancing (Section 6.3).

4.2 Low Usage of Network and I/O

In addition to compute resources, network and I/O are also frequently used in distributed ML. To understand their impact,
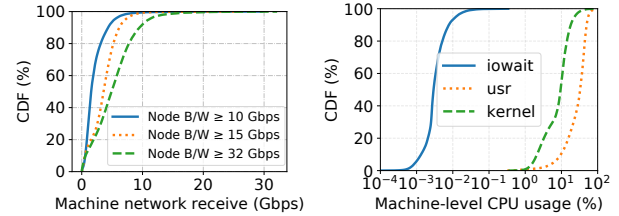
(a) CDF of machine network input. (b) CDF of machine CPU time.

Figure 7: Low usage of network and I/O.

we measure the network input rate[3] in machines with different bandwidth guarantees (≥ 10 Gbps for P100 and Misc, ≥ 15 Gbps for T4, and ≥ 32 Gbps for V100) and depict their distributions in Figure 7a. The P95 network input rate only reaches 54%, 48%, and 34% of the guaranteed bandwidth provided in P100 (or Misc), T4, and V100 machines, respectively.

In terms of I/O, we collect machine-level CPU usage data, including the I/O waiting time (iowait) and the execution time in usr and kernel modes, respectively. Figure 7b shows their distributions. The CPU time spent on iowait is three orders of magnitude smaller than that in usr and kernel modes, meaning that CPUs are mostly busy processing data rather than waiting for the I/O to complete.

5 Opportunities for Cluster Management

In PAI, our goal of cluster management is two-fold: (1) achieving high utilization in GPU machines, and (2) completing as many tasks as fast as possible. In this section, we describe the opportunities and our efforts in achieving the two goals.

5.1 GPU Sharing

Unlike CPUs, GPUs do not natively support sharing and are allocated as indivisible resources in many production clusters [36, 72], where a single task instance runs exclusively on a GPU. Although such allocation provides strong performance isolation, it results in GPU underutilization, which is particularly salient in our cluster as most instances can only utilize a small portion of the allocated GPUs (Section 3.3).

To avoid this problem, the PAI cluster scheduler supports GPU sharing which allows multiple task instances to run on the same GPU in a space- and time-multiplexed manner. With this feature, a task instance can request a fraction of GPU (< 1 GPU) and is guaranteed to allocate the specified fraction of GPU memory upon scheduling (space-multiplexed). When needed, an instance can also use unallocated GPU memory during execution. An instance, however, has no guaranteed allocation of compute units (i.e., SMs), which are dynamically shared among co-located instances (time-multiplexed).[4]

---

[3]Our trace does not log the network output. For most training and inference tasks, the network input is orders of magnitude larger than the output.

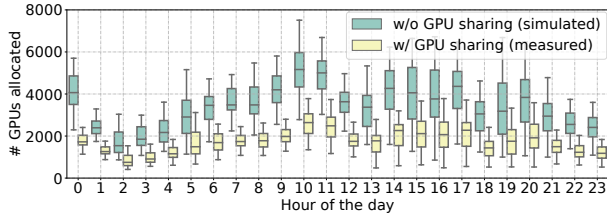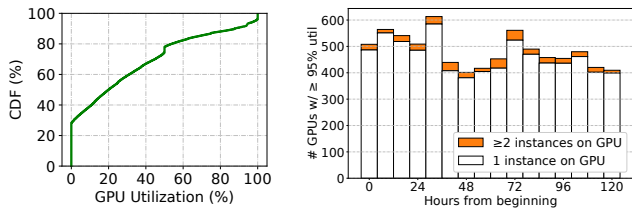[4]Fine-grained sharing of compute units with isolation guarantee requires

Figure 8: Box plots of the number of allocated GPUs with and without GPU sharing. The boxes depict the 25th, 50th, and 75th percentiles, respectively; the two whiskers are one interquartile range (IQR) past the low and high quartiles.

(a) CDF of GPU utilization.

(b) The number of GPUs with ≥ 95% utilization in a 5-day period. The top-stacked bars (orange) show GPUs running ≥ 2 instances.

Figure 9: Heavy utilization is rarely measured in GPUs; most heavy-utilized GPUs run a single instance.

Benefits of GPU sharing. GPU sharing enables considerable savings on resource provisioning. To see this, we simulate the scenario of no GPU sharing, in which we replay the trace and count the number of allocated GPUs in each hour. Figure 8 compares the simulated results with the numbers measured in the real system, binned in hour of the day. On average, only 50% of GPUs are needed with sharing. In the peak hour at around 10 am, the savings can be up to 73%.

Does GPU sharing cause contention? As the utilization increases, instances running on a shared GPU start to contend for streaming processors (SMs), causing interference. To quantify how frequently the contention may occur, we collect the utilization data of all GPUs in two months and depict their distribution in Figure 9a. Heavy utilization (≥ 95%) is rarely measured, which accounts for only 7% cases in the trace. We further examine those heavy-utilized GPUs in which running instances have a high chance to contend with each other. Figure 9b shows the number of heavy-utilized GPUs in a 5-day period, among which only a few (4.5% on average) run multiple instances (the top-stacked bars). As the majority of heavy-utilized GPUs run a single instance, no contention occurs. We therefore believe GPU sharing does not cause severe contention in our cluster.

---

high-level support of ML framework. In PAI, such support is provided by AntMan [66]. Yet, it only applies to tasks running in the frameworks where AntMan is implemented (currently supporting TensorFlow and PyTorch).
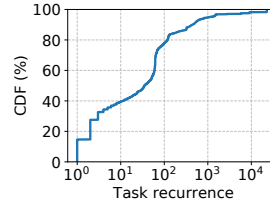

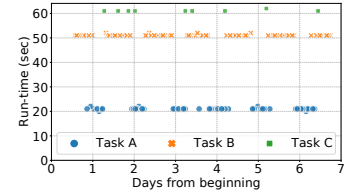Figure 10: CDF of task recurrence.

Figure 11: Submissions and instance run-times of three batch inference tasks using BERT.

5.2 Predictable Duration for Recurring Tasks

Knowing the duration (aka run-time) of ML task instances is the key to making better scheduling decisions. Existing schedulers for ML workloads predict the task instances duration based on the training progress (e.g., number of iterations, loss curve, and target accuracy) and speed of the task [29, 41, 46, 49]. Obtaining such information requires specific framework support (e.g., TensorFlow and PyTorch), which is not always possible in our cluster as users run a variety of frameworks of standard or customized version, and their submitted tasks may not perform iterative training (e.g., inference). In fact, our cluster scheduler [26, 71] is designed for container workloads and is agnostic to the task semantics.

The prevalence of recurring tasks. Despite the scheduler being agnostic to task progress, we find that most tasks are *recurring*, and their instance run-times can be well predicted from past executions. Yet, in our system, task recurrence cannot be simply identified from the task ID or name, which is uniquely generated for each submission. Instead, we turn to the meta-information consistently specified by a task across multiple submissions, such as the entry scripts, command-line parameters, data sources and sinks. Hashing the meta-information generates a unique Group tag, which we use to identify the recurrence of a task. Following this approach, we depict the distribution of task recurrences in Figure 10: around 65% tasks repeatedly run at least 5 times in the trace.

In addition to periodic training, many recurring tasks perform *batch inference*. These tasks aggregate data from incoming requests and then perform batch inference on a collective of data in one go. Users can configure the task launching interval, ranging from minutes to days. As an illustrative example, Figure 11 shows three recurring tasks identified in the trace that perform batch inference with pre-trained BERT [23] models. All three tasks run on a regular basis, with stable average instance run-times that can be accurately predicted.

Instance duration prediction for recurring tasks. A recurring task can be submitted by different users with different resource requests, and its instances may have different run-times. We therefore predict the duration from past runs based on three features, the task's username (User), resource requests (Resource, including GPU and other resources), and group tag (Group). Taking these features as input, we predict
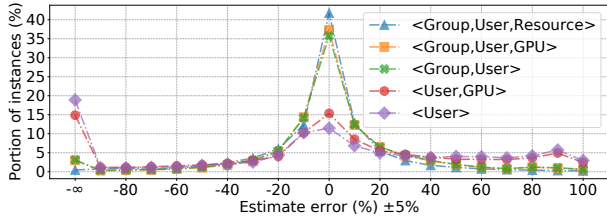
Figure 12: Percentage prediction error, i.e., $(true-pred)/_{true}$ in percentage, of duration estimates with different features.
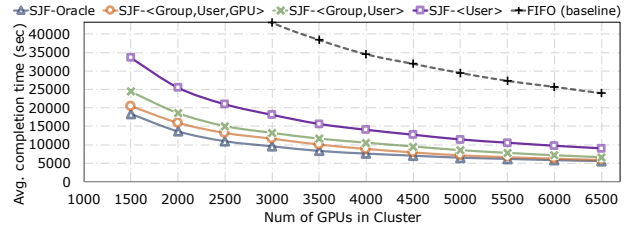

Figure 13: Average task completion time given different GPU cluster sizes and various scheduling policies in simulation.

These results are in line with Figure 12.

6 Challenges of Scheduling

Compared to previous simulations, scheduling ML tasks of large heterogeneity in production clusters is far more complex. To understand the challenges posed by such heterogeneity, in this section we present case studies for two representative types of ML tasks with high and low GPU requests. We describe our scheduling policies deployed in production that differentiate between the two types of tasks in light of their different request and usage patterns. Yet, many challenges remain open, which we discuss in detail.

6.1 Case Study of High-GPU Tasks

In our cluster, a small portion of tasks run compute-intensive instances with high GPU requests (Section 3.3). These tasks train state-of-the-art models or perform inference with trained models for business-critical, user-facing applications. They request powerful GPU devices with high memory or advanced hardware features (e.g., NVLink).

NLP with advanced language models. Around 6.4% tasks running in our cluster perform natural language processing (NLP) using advanced models, such as BERT [23], AL-BERT [37], and XLNet [67]. Among them, 73% have large input and must run on GPUs with 16 GiB or higher memory (i.e., T4, P100, V100/V100M32). Figure 14a shows the distribution of GPU requests and usages of NLP instances, where 40% request more than 1 GPU and use over 0.4 GPUs in computing power. Comparing Figure 5b and Figure 14a, we observe much higher GPU requests and usages of NLP tasks than that of general workloads.

Image classification with massive output. In our cluster, some distributed training tasks request to run their worker instances in one machine with high-speed GPU-to-GPU inter-connects (e.g., NVLink) for much improved performance, a requirement known as *GPU locality*. A typical example is to train a classification model that classifies images of goods into a large number of standard product units (SPUs). The model can be a modified ResNet [33] with the last output layer replaced by a softmax layer with 100,000 output of SPUs

the task's average instances duration using the CART (Classification And Regression Trees [17]) algorithm with a tree regressor. The regressor makes at most 10 splits for each tree and uses the mean absolute error (MAE) as the splitting criterion. We choose MAE instead of the standard mean squared error (MSE) because the former is more robust to extreme outliers in heavy-tailed distribution than the latter.

To evaluate the accuracy of our prediction, we consider tasks that recur at least 5 times in the trace. We use 80% of those tasks to train the predictor and the remaining 20% for testing. Figure 12 compares the accuracy of the predictor trained with different feature inputs, including Group, User, Resource, and GPU (requested GPU types and numbers). We use percentage prediction error [35] as the accuracy metric, defined as $(true-pred)/true \times 100\%$. Our evaluation shows that Group is the most important feature that greatly improves the prediction accuracy. Further complementing it with User and Resource (or GPU) results in less than 25% prediction error for 78% instances. According to prior studies [16], duration predictions with such accuracy is sufficient to make high-quality scheduling decisions.

Benefits for scheduling. We present a simple simulation study to evaluate how the prediction of task instance duration can help improve scheduling. We developed a discrete-time simulator and use it to replay the trace. We sample tasks from the trace and feed their resource requests, arrival times, real and predicted run-times into the simulator. We assume homogeneous GPUs in simulation and respect the real duration when scheduling a task instance to a GPU. Both the simulator and experiment scripts are released along with the trace [1].

We configure two scheduling policies, first-in-first-out (FIFO) and shortest-job-first (SJF), in simulation. Figure 13 shows the average task completion time in GPU clusters of different sizes using FIFO and four SJF schedulers, where SJF-Oracle makes scheduling decisions based on the real-measured task instance duration (ground truth) and the others use predictors trained with different input features. Compared to FIFO, the four SJF schedulers reduce the average task completion time by 63–77%, depending on the predictors they use. In particular, the predictors trained with the Group feature yield better performance; the more features are included, the more accurate the predictions are, and the closer the scheduling performance is to the optimum (SJF-Oracle).

(a) CDF of GPU requests and us-
ages of NLP task instances.

(b) Per-epoch duration of 3 classifi-
cation models trained in 8-GPU ma-
chines with and without NVLink.

Figure 14: High-GPU tasks (NLP and image classification).

(a) CPU usage of instances.

(b) GPU usage of instances.

Figure 15: CDF of the CPU and GPU usage of click-through-
rate (CTR) instances.

(a) Duration breakdown of CTR pre-
diction instances.

(b) DeepFM training instances in-
terfered by the co-located load.

Figure 16: Microbenchmark of inference and training in-
stances of click-through-rate prediction models.

(a) CDF of CPU and GPU usage.

(b) Instance duration breakdown.

Figure 17: Resource usage and duration breakdown of GNN
training instances.

(ResNet-100k). The presence of such a large fully-connected
layer mandates the exchange of massive gradient updates be-
tween worker instances, making communication a bottleneck.
For these tasks, meeting GPU locality is critically important.
Figure 14b compares the duration of a training epoch of three
classification models with a large number of output in 8-GPU
machines with and without NVLink (i.e., via PCIe). All three
models achieve salient speedup with NVLink: ResNet-100k,
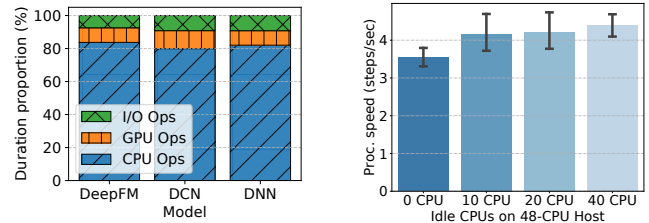the largest model, is accelerated by 10.5×.
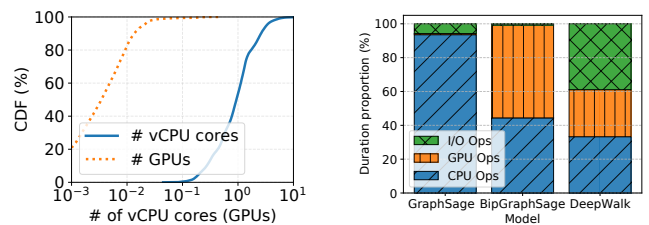
6.2 Case Study of Low-GPU Tasks

The majority of tasks running in our cluster have low GPU
requests and usages (Section 3.3). To understand this some-
what unexpected result, we study three popular tasks. By
profiling their executions, we find that they spend a consider-
able amount of time on CPUs for data processing (e.g., data
fetching, feature extraction, sampling) and simulation (e.g.,
reinforcement learning), leaving GPUs under-utilized.

CTR prediction model training and inference. Among
all tasks in the trace, over 6.7% are for advertisement click-
through rate (CTR) prediction. These tasks use a variety of
CTR models [30, 60, 73, 74], with around 25% instances per-
forming training and the other 75% performing inference. Fig-
ure 15 shows the distributions of the CPU and GPU usages
of these instances. Compared to training, inference instances
have higher CPU utilization as they process a large volume
of data continuously arriving. Both instances have low GPU
utilization: over 75% instances use less than 0.1 GPUs.

We next profile the executions of three inference instances
with DeepFM, DCN, and DNN models, respectively. Fig-
ure 16a shows the run-time breakdown of I/O, GPU, and
CPU operations. The three instances spend around 80% run-
time on CPUs to fetch and process the next input batch
(IteratorGetNext in TensorFlow [20, 40]); GPU and I/O opera-
tions (e.g., MatMul, Sum, Cast, MEMCPYHtoD) only account
for 10% of the execution time, respectively.

The high CPU usage of these instances makes them prone
to interference from the co-located workload, especially in
machines with high CPU utilization. To see this, we run train-
ing instances of a DeepFM model in containers with 8 vCPU
cores. Together with an instance, we run some artificial load
using spare cores of the host machine to create CPU stress. We
configure varying load to control the level of stress. Figure 16b
shows the instance training speed in a 48-core machine under
varying stresses with 0 to 40 cores left idle (highest to no
stress). Though the co-located load run on different vCPU
cores not occupied by the instance, it still results in up to
28% slowdown of the training speed due to the contention
of other shared resources, such as cache, power, and memory
bandwidth [19, 21, 58].

GNN training. Graph Neural Network (GNN) training
comes as another popular computation, which accounts for
2% instances in our cluster, including GraphSage [31], Bi-
partite GraphSage [75], GAT [57], etc. Figure 17a shows
the distribution of CPU and GPU usage of GNN training in-
stances, where CPU is more heavily utilized than GPU. In
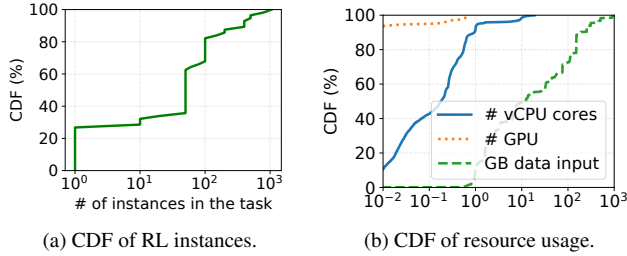production GNN models, a graph must undergo a sequence

(a) CDF of RL instances. (b) CDF of resource usage.

Figure 18: Characterization of reinforcement learning instances.

(a) Queueing delays of all instances and tasks. (b) Queueing delays of V100 instances and tasks.

Figure 19: Task queueing delays in simulation with load-balancing (Balanced) and reserving-and-packing (R&P).

of pre-processing, such as EdgeIteration, NeighborSampling, and NegativeSampling [75], before turning into an embedding (a computationally digestible format, usually vectors) of a deep neural network. Such graph pre-processing is currently cost-effective when performing on CPUs. As shown in Figure 17b, it accounts for 30–90% duration of each training iteration in different models.

Reinforcement learning. Our cluster also runs many reinforcement learning (RL) tasks. An RL algorithm iteratively generates a batch of data through parallel simulations on CPUs and performs training with the generated data on GPUs to improve the learning policy. Figure 18a shows that 72% RL tasks have at least 10 gang-scheduled instances, with the largest one running over 1,000 instances. Most RL instances are used to run simulations, eating up lots of CPUs and network bandwidth but only a small fraction of GPUs, as shown in Figure 18b. In fact, in the largest RL task, each instance requests only 0.05 GPUs.

6.3 Deployed Scheduling Policies

Compared to low-GPU tasks, high-GPU tasks have picky scheduling requirements and are usually run by business-critical applications. They are hence differentiated from other tasks and scheduled as first-class citizens.

Reserving-and-packing. In our cluster, the scheduler employs a *reserving-and-packing* policy. That is, it intentionally reserves high-end GPUs (e.g., V100/V100M32 with NVLinks) for high-GPU tasks, while packing the other workloads to machines with less advanced GPUs (e.g., T4 and Misc). Specifically, for each task, the scheduler characterizes its *computation efficiency* using a performance model that accounts for many task features, such as the degree of parallelism, the used ML model, the size of embedding [59, 64, 70], and the historical profiles of other similar tasks. Tasks with high computation efficiency larger than a certain threshold are identified as high-GPU.

For each task, the scheduler generates an ordered sequence of *allocation plans*; each plan specifies the intended GPU device and is associated with an attempt timeout value. The scheduler attempts allocation following the ordered plans: it waits for the availability of the intended GPU specified in the
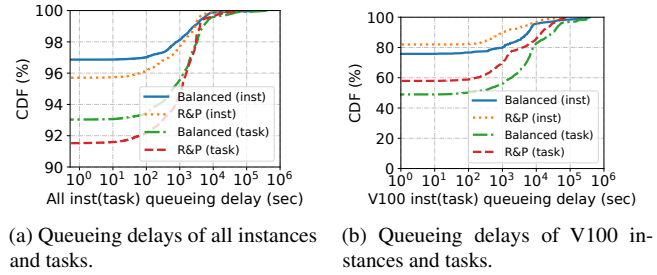
current plan until timeout, and then moves on to the next plan for another attempt. For high-GPU tasks, the allocations of high-end GPUs are attempted before the less advanced ones in the ordered plans; for other tasks, the order is reversed. Our GPU scheduler is implemented atop Fuxi [26, 71], a locality-tree based scheduling system.

Load-balancing. Given the potential resource contention and interference between co-located task instances (Section 6.2), maintaining load balancing across machines with similar specs is also important. Therefore, under reserving-and-packing, the scheduler also prioritizes instance scheduling to machines with low *allocation rate*, measured as a weighted sum of the allocated CPUs, memory, and GPUs normalized by the machine's capacity.

Benefits. Our scheduler prioritizes reserving-and-packing over load-balancing. To justify this design, we evaluate two scheduling policies using the simulator described in Section 5.2: ① simply load-balancing machines using progressive filling (always scheduling a task's instances to the least utilized node), and ② only performing reserving-and-packing without considering load balancing (R&P). We sample 100,000 tasks with over 500,000 gang-scheduled instances from the trace and feed them into the simulator. Figure 19a shows the CDF of the queueing delays of all instances and tasks under the two policies. Note that the queueing delay of a task is also the queueing delay of its gang-scheduled instances. Over 90% instances and tasks are launched immediately under the two policies. Compared to load-balancing, reserving-and-packing reduces the average task queueing by 45%, mostly attributed to the significant cutoff of the tail latency by over 10,000 seconds. Figure 19b further compares the queueing delays of business-critical tasks and instances requesting V100 GPUs under the two policies: reserving-and-packing reduces the average task queueing delay by 68%. The simulation results justify our design of prioritizing reserving-and-packing over load-balancing.

6.4 Open Challenges

However, our scheduler policy design is not without its problems, many of which remain open to address. We next discuss

Table 2: Mismatch between machine specs and instance requests, in terms of the provisioned/requested CPUs per GPU.

vCPU cores per GPU	All nodes	8-GPU nodes	2-GPU nodes
Machine specs	23.2	12.0	38.1
Instance requests	21.4	22.8	18.1

those open challenges, which we believe also stand in other GPU clusters with heterogeneous machines.

Mismatch between machine specs and instance requests.
We observe a mismatch between machine specs and instance requests. Table 2 compares the average number of provisioned and requested vCPU cores per GPU in machines with 8 and 2 GPUs and their running instances. In 8-GPU machines, 12 vCPU cores are provisioned for each GPU. Yet, the instances running in those machines request 22.8 vCPU cores per GPU on average. On the other hand, CPUs in 2-GPU machines are over-provisioned, where the CPU-to-GPU ratio is more than twice of the instance requests.

To understand how the mismatch may affect the machine utilization, we randomly sample a number of nodes with different specs and depict the requests and usages of CPUs and GPUs in heatmaps shown in Figure 20, where each row corresponds to one machine, and all values are normalized to the machine's capacity. Compared to 8-GPU nodes, 2-GPU machines have substantially underutilized CPUs despite GPUs being heavily occupied. On average, P100 (T4) machines have 31% (20%) CPUs allocated with only 19% (10%) CPU utilization (Figures 20c and 20d).

We stress that the mismatch between machine specs and instance requests is not fundamental, as the cluster-wide CPU-to-GPU specs remains close to the overall instance requests (23.2 vs. 21.4 as shown in Table 2). We therefore believe that the mismatch can be avoided or at least mitigated by improved scheduling (e.g., rescheduling some high-CPU instances in 8-GPU machines to 2-GPU nodes).

Overcrowded weak-GPU machines. Compared to other machines, those with less advanced GPUs are overcrowded. The problem becomes even more salient in 8-GPU nodes (Misc GPUs) as shown in Figure 20a. On average, 77% CPUs and 74% GPUs are allocated in these machines. CPUs are better utilized than GPUs: the utilization of CPU is 43% on average, while the average utilization of GPU is 18%. This result is partly caused by our scheduling algorithm prioritizing weak-GPU machines for low-GPU tasks (Section 6.3), which account for a large instance population in our cluster.

Imbalanced load in high-end machines. Compared to other nodes, high-end machines with advanced V100 GPUs are less crowded (Figure 20b), with the average allocation ratios of CPUs and GPUs being 35% and 49%, respectively. These machines are usually reserved for a small number of important high-GPU tasks, thus suffering from low utilization. We also observe imbalanced load among V100 machines. In

(a) Machines with 8 Misc GPUs.   (b) Machines with 8 V100 GPUs.

(c) Machines with 2 P100 GPUs.   (d) Machines with 2 T4 GPUs.
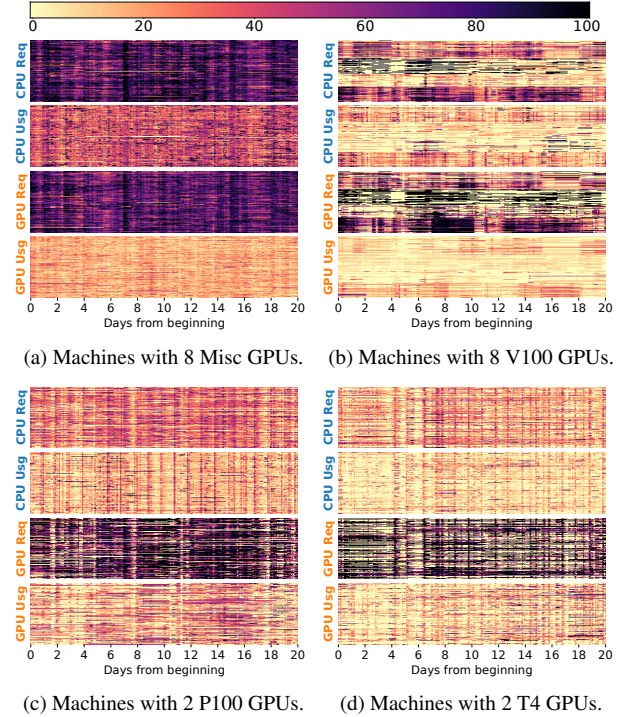
Figure 20: Heatmap of requests and usages of CPU and GPU in machines with different specs. Each row corresponds to one machine.

Figure 20b, the machines near the bottom are more crowded than the others. This suggests that the current load-balancing algorithm still has plenty room to improve (Section 6.3).

CPU can be the bottleneck. As shown in Section 6.2, a large number of ML tasks use CPUs more extensively than GPUs. These tasks are more likely to get slowdown in machines with high CPU contentions. To see this, we study the correlation between machine utilization and instance slowdown in the trace and depict the results in Figure 21. Our analysis focuses on the recurring tasks (Section 5.2). In each task recurrence, we divide the instances into three groups: 1) instances with *accelerated* execution whose duration is the shortest 15%, 2) *normal* execution whose duration is the middle 70%, and 3) *delayed* execution whose duration is the longest 15%. Figure 21a compares the CPU utilization in machines running accelerated, normal, and delayed instances. In general, machines running delayed instances measure higher CPU utilization than those running accelerated and normal instances. However, such correlation is not found on GPUs. As illustrated in Figure 21b, the distributions of GPU utilization show no substantial differences across machines running accelerated, normal, and delayed instances.

We next zoom in to the popular CTR prediction tasks with high CPU usage (Section 6.2). Figure 22 shows the CDF of CPU/GPU utilization in machines running accelerated and delayed instances, respectively. In machines with over 24% CPU utilization run 50% delayed instances but only 10%
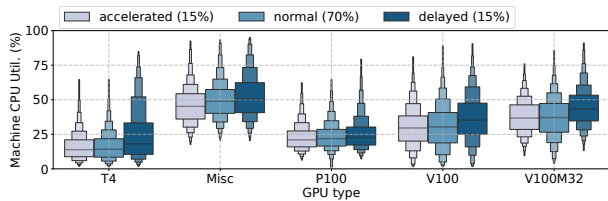
(a) CPU utilization of machines with various GPU types.
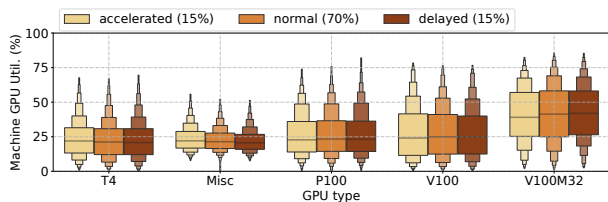
(b) GPU utilization of machines with various GPU types.

Figure 21: Correlation between machine utilization (CPU and GPU) and instance slowdown. Machines hosting delayed instances have higher CPU utilization than those hosting normal and accelerated ones. In contrast, such correlation is not found on GPUs. The boxes depict the $1/128, 1/64, \ldots, 1/4, 1/2, 3/4, \ldots, 63/64, 127/128$ quantile values [34, 61].
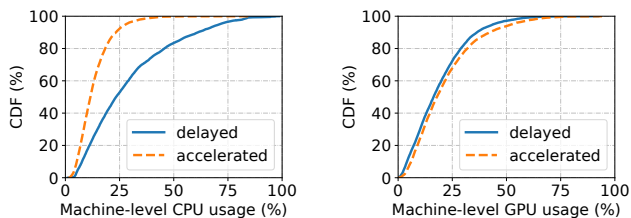
(a) CDF of machine CPU usage.   (b) CDF of machine GPU usage.

Figure 22: The impact of resource utilization to the execution of CTR prediction instances with high-CPU usages.

accelerated instances (Figure 22a), an evidence of strong correlation between CPU contention and instance slowdown. GPU contention, on the other hand, has no clear contribution to instance slowdown (Figure 22b).

To summarize, task instance scheduling in GPU clusters should also account for the potential interference caused by CPU contentions. This essentially calls for a multi-resource scheduler that jointly considers CPUs, GPUs, memory, I/O, and network when making scheduling decisions.

7 Discussion

Support of elastic scheduling. One fundamental challenge posed to GPU schedulers in heterogeneous clusters is the gang-scheduling requirement of distributed training. Some frameworks [6, 13] are hence developed to support elastic scheduling which allows a training job to dynamically adjust the number of workers on the fly. Compared to gang-

scheduling jobs, elastic-scheduling jobs are easier to handle: they can start with a small amount of resources and later scale to more GPUs when the cluster becomes less crowded. However, elastic scheduling introduces non-determinism to final model accuracy [27, 63].

Machine provisioning and resource disaggregation. GPU schedulers should also account for machine provisioning: in our previous analysis, although 8-GPU machines provide abundant GPU processing power, 2-GPU machines can be a better fit to tasks with heavy CPU processing. To make the problem simplified, many system works propose to decompose monolithic machines into a number of distributed, disaggregated hardware components for improved hardware elasticity [53], despite the non-negligible communication overhead. TensorFlow has recently made a framework-level attempt towards this direction. It released an experimental data service [5] to decouple data pre-processing from GPU training so as to address the CPU bottleneck. However, it requires changing user's source code with non-trivial efforts.

8 Related Work

GPU sharing. GPU sharing can be supported at different levels. At the GPU hardware level, NVIDIA recently released the Multi-Instance GPU (MIG) [10] feature that enables partitioning a large GPU into multiple small GPU instances with isolated memory and bandwidth. However, MIG is only available on the latest A100 GPUs, and it does not support arbitrary GPU partition. At the GPU software level, GPU time-multiplexing can be implemented by intercepting CUDA APIs [24, 28, 54]. Yet, it usually introduces non-trivial context switching overhead and does not provide a good isolation between the co-located task instances. NVIDIA Multi-Process Service (MPS) [11] offers an alternative solution, but it cannot isolate failures among co-executed process. At the framework level, by extending standard ML frameworks such as TensorFlow and PyTorch, AntMan [66] and Salus [68] enable fine-grained GPU sharing and manage GPU memory for each task instance at a low cost. However, Salus requires users to adapt their code to the framework, while AntMan only supports training tasks.

GPU cluster scheduler. Many GPU cluster schedulers have been proposed recently (Table 1). Notably, Optimus [49] and Tiresias [29] schedule distributed training jobs with an objective of minimizing the average completion time; Themis [41], Gandiva$_{fair}$ [18], and HiveD [72] further consider completion-time fairness for the training jobs. All these works support no GPU sharing, with the minimum allocation unit being one GPU. The clusters used in evaluation are of limited size, workload diversity, and machine heterogeneity.

ML workload characterization. In addition to computation, communication and I/O are also important for distributed training and are thus the focus in the previous characterization

studies. For example, `tf.data` [45] reports that a majority of production ML workloads read many terabytes of data and spend a large proportion of time in data loading. Some ML schedulers [50, 52] study the training efficiency with different network bandwidth and propose to mitigate the communication overhead for accelerated training. An earlier characterization of ML training tasks in Alibaba PAI [59] suggests to replace the PS-Worker architecture with Ring AllReduce to better exploit the high-speed NVLink among GPUs. These works mainly focus on distributed training but leave aside the general MLaaS workloads and cluster resource management.

9   Conclusion

In this paper, we characterized a two-month production trace consisting of a mix of training and inference tasks in a large GPU cluster of Alibaba PAI. We made a number of observations. Notably, the majority of tasks have gang-scheduled instances and are executed recurrently. Most of them are small, requesting less than one GPU per instance, whereas a small number of business-critical tasks demand high-end GPUs interconnected by NVLinks in one machine. For those low-GPU tasks, CPU is often the bottleneck, which is used for data pre-processing and simulation. To better schedule the PAI workloads, our scheduler enables GPU sharing and employs a reserving-and-packing policy that differentiates the high-GPU tasks from the low-GPU ones. We also identified a few challenges that remain open to address, including load imbalance in heterogeneous machines and the potential CPU bottleneck. We have released the trace to facilitate future research on improved GPU scheduling.

10   Acknowledgment

References

[1] Alibaba cluster trace program. https://github.com/alibaba/clusterdata, 2021.

[2] Alibaba machine learning platform for AI. https://www.alibabacloud.com/product/machine-learning, 2021.

[3] Amazon machine learning. https://docs.aws.amazon.com/machine-learning, 2021.

[4] Azure AI. https://azure.microsoft.com/en-us/overview/ai-platform/, 2021.

[5] Distributed tf.data service. https://github.com/tensorflow/community/blob/master/rfcs/20200113-tf-data-service.md, 2021.

[6] ElasticDL: A Kubernetes-native deep learning framework. https://github.com/sql-machine-learning/elasticdl, 2021.

[7] Google Cloud Vertex AI. https://cloud.google.com/vertex-ai, 2021.

[8] IBM Waston. https://www.ibm.com/watson, 2021.

[9] NVIDIA Management Library (NVML). https://developer.nvidia.com/nvidia-management-library-nvml, 2021.

[10] NVIDIA Multi-Instance GPU (MIG) user guide. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/, 2021.

[11] NVIDIA Multi-Process Service (MPS). https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf, 2021.

[12] NVIDIA NVLink and NVSwitch. https://www.nvidia.com/en-us/data-center/nvlink/, 2021.

[13] TorchElastic. https://pytorch.org/elastic, 2021.

[14] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A system for large-scale machine learning. In *Proc. USENIX OSDI*, 2016.

[15] Marcelo Amaral, Jordà Polo, David Carrera, Seetharami Seelam, and Malgorzata Steinder. Topology-aware GPU scheduling for learning workloads in cloud environments. In *Proc. ACM/IEEE SC*, 2017.

[16] George Amvrosiadis, Jun Woo Park, Gregory R Ganger, Garth A Gibson, Elisabeth Baseman, and Nathan DeBardeleben. On the diversity of cluster workloads and its impact on research results. In *Proc. USENIX ATC*, 2018.

[17] Leo Breiman, Jerome Friedman, Charles J Stone, and Richard A Olshen. *Classification and regression trees*. CRC press, 1984.

[18] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. Balancing efficiency and fairness in heterogeneous GPU clusters for deep learning. In *Proc. ACM EuroSys*, 2020.

[19] Shuang Chen, Christina Delimitrou, and José F Martínez. Parties: QoS-aware resource partitioning for multiple interactive services. In *Proc. ASPLOS*, 2019.

[20] Maxwell Collard. TensorFlow performance bottleneck on IteratorGetNext. https://stackoverflow.com/q/48715062, 2021.

[21] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Proc. ASPLOS*, 2013.

[22] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *Proc. IEEE CVPR*, 2009.

[23] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[24] José Duato, Antonio J Pena, Federico Silla, Rafael Mayo, and Enrique S Quintana-Ortí. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *Proc. HPCS*, 2010.

[25] Raj Dutt, Torkel Ödegaard, and Anthony Woods. Grafana: The open observability platform. https://grafana.com/, 2021.

[26] Yihui Feng, Zhi Liu, Yunjian Zhao, Tatiana Jin, Yidi Wu, Yang Zhang, James Cheng, Chao Li, and Tao Guan. Scaling large production clusters with partitioned synchronization. In *Proc. USENIX ATC*, 2021.

[27] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: Training ImageNet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[28] Jing Gu, Shengbo Song, Ying Li, and Hanmei Luo. GaiaGPU: sharing GPUs in container clouds. In *ISPA/IUCC/BDCloud/SocialCom/SustainCom*, 2018.

[29] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *Proc. USENIX NSDI*, 2019.

[30] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. DeepFM: a factorization-machine based neural network for CTR prediction. *arXiv preprint arXiv:1703.04247*, 2017.

[31] William L Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. *arXiv preprint arXiv:1706.02216*, 2017.

[32] Tomoki Hayashi, Ryuichi Yamamoto, Katsuki Inoue, Takenori Yoshimura, Shinji Watanabe, Tomoki Toda, Kazuya Takeda, Yu Zhang, and Xu Tan. Espnet-TTS: Unified, reproducible, and integratable open source end-to-end text-to-speech toolkit. In *Proc. IEEE ICASSP*, 2020.

[33] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proc. IEEE CVPR*, pages 770–778, 2016.

[34] Heike Hofmann, Hadley Wickham, and Karen Kafadar. Value plots: Boxplots for large data. *J. Comput. Graph. Stat.*, 26(3):469–477, 2017.

[35] Rob J Hyndman and George Athanasopoulos. *Forecasting: principles and practice*. OTexts, 2018.

[36] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant GPU clusters for DNN training workloads. In *Proc. USENIX ATC*, 2019. https://github.com/msr-fiddle/philly-traces.

[37] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. ALBERT: A lite BERT for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942*, 2019.

[38] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Joseph Gonzalez, Ken Goldberg, and Ion Stoica. Ray RLlib: A composable and scalable reinforcement learning library. *arXiv preprint arXiv:1712.09381*, 2017.

[39] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[40] Chi Keung Luk, Jose Americo Baiocchi Paredes, Russell Power, and Mehmet Deveci. Debugging correctness issues in training machine learning models, November 12 2020. US Patent App. 16/403,884.

[41] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient GPU cluster scheduling. In *Proc. USENIX NSDI*, 2020.

[42] Deborah T Marr, Frank Binns, David L Hill, Glenn Hinton, David A Koufaty, J Alan Miller, and Michael Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technol. J.*, 6(1), 2002.

[43] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proc. ICML*, 2016.

[44] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2015.

[45] Derek G Murray, Jiri Simsa, Ana Klimovic, and Ihor Indyk. tf. data: A machine learning data processing framework. *arXiv preprint arXiv:2101.12127*, 2021.

[46] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-aware cluster scheduling policies for deep learning workloads. In *Proc. USENIX OSDI*, 2020.

[47] Jongsoo Park, Maxim Naumov, Protonu Basu, Summer Deng, Aravind Kalaiah, Daya Khudia, James Law, Parth Malani, Andrey Malevich, Satish Nadathur, et al. Deep learning inference in Facebook data centers: Characterization, performance optimizations and hardware implications. *arXiv preprint arXiv:1811.09886*, 2018.

[48] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An imperative style, high-performance deep learning library. In *Proc. NeurIPS*, 2019.

[49] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proc. ACM EuroSys*, 2018.

[50] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed DNN training acceleration. In *Proc. ACM SOSP*, 2019.

[51] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, et al. MLPerf inference benchmark. In *Proc. ACM/IEEE ISCA*, 2020.

[52] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling distributed machine learning with in-network aggregation. In *Proc. USENIX NSDI*, 2021.

[53] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *Proc. USENIX OSDI*, 2018.

[54] Lin Shi, Hao Chen, Jianhua Sun, and Kenli Li. vCUDA: GPU-accelerated high-performance computing in virtual machines. *IEEE Trans. Comput.*, 61(6):804–816, 2011.

[55] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proc. IEEE CVPR*, 2016.

[56] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proc. NIPS*, 2017.

[57] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[58] Luping Wang, Qizhen Weng, Wei Wang, Chen Chen, and Bo Li. Metis: learning to schedule long-running applications in shared container clusters at scale. In *Proc. ACM/IEEE SC*, 2020.

[59] Mengdi Wang, Chen Meng, Guoping Long, Chuan Wu, Jun Yang, Wei Lin, and Yangqing Jia. Characterizing deep learning training workloads on Alibaba-PAI. In *Proc. IEEE IISWC*, 2019.

[60] Ruoxi Wang, Bin Fu, Gang Fu, and Mingliang Wang. Deep & cross network for ad click predictions. In *Proc. ACM ADKDD*, 2017.

[61] Michael L Waskom. Seaborn: statistical data visualization. *J. Open Source Softw.*, 6(60):3021, 2021.

[62] Shinji Watanabe, Takaaki Hori, Shigeki Karita, Tomoki Hayashi, Jiro Nishitoba, Yuya Unno, Nelson Enrique Yalta Soplin, Jahn Heymann, Matthew Wiesner, Nanxin Chen, Adithya Renduchintala, and Tsubasa Ochiai. ESPnet: End-to-end speech processing toolkit. In *Proc. INTERSPEECH*, 2018.

[63] Pijika Watcharapichat, Victoria Lopez Morales, Raul Castro Fernandez, and Peter Pietzuch. Ako: Decentralised deep learning with partial gradient exchange. In *Proc. ACM SoCC*, 2016.

[64] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, 2009.

[65] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective cluster scheduling for deep learning. In *Proc. USENIX OSDI*, 2018.

[66] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li Li, Yihui Feng, Wei Lin, and Yangqing Jia. AntMan: Dynamic scaling on GPU cluster for deep learning. In *Proc. USENIX OSDI*, 2020.

[67] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V Le. XLNet: Generalized autoregressive pretraining for language understanding. *arXiv preprint arXiv:1906.08237*, 2019.

[68] Peifeng Yu and Mosharaf Chowdhury. Salus: Fine-grained GPU sharing primitives for deep learning applications. In *Proc. MLSys*, 2020.

[69] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. Mark: Exploiting cloud services for cost-effective, SLO-aware machine learning inference serving. In *Proc. USENIX ATC*, 2019.

[70] Wei Zhang, Wei Wei, Lingjie Xu, Lingling Jin, and Cheng Li. AI Matrix: A deep learning benchmark for Alibaba data centers. *arXiv preprint arXiv:1909.10562*, 2019.

[71] Zhuo Zhang, Chao Li, Yangyu Tao, Renyu Yang, Hong Tang, and Jie Xu. Fuxi: a fault-tolerant resource management and job scheduling system at internet scale. In *Proc. VLDB Endowment*, 2014.

[72] Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Fan Yang, Lidong Zhou, Mao Yang, Francis CM Lau, Yuqi Wang, Yifan Xiong, et al. HiveD: Sharing a GPU cluster for deep learning with guarantees. In *Proc. USENIX OSDI*, 2020.

[73] Guorui Zhou, Na Mou, Ying Fan, Qi Pi, Weijie Bian, Chang Zhou, Xiaoqiang Zhu, and Kun Gai. Deep interest evolution network for click-through rate prediction. In *Proc. AAAI*, 2019.

[74] Guorui Zhou, Xiaoqiang Zhu, Chenru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. Deep interest network for click-through rate prediction. In *Proc. ACM KDD*, 2018.

[75] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. AliGraph: a comprehensive graph neural network platform. In *Proc. VLDB Endowment*, 2019.

Evolvable Network Telemetry at Facebook

Yang Zhou[†]   Ying Zhang[‡]   Minlan Yu[†]   Guangyu Wang[‡]   Dexter Cao[‡]   Eric Sung[‡]   Starsky Wong[‡]

[†]*Harvard University*   [‡]*Facebook*

Abstract

Network telemetry is essential for service availability and performance in large-scale production environments. While there is recent advent in novel measurement primitives and algorithms for network telemetry, a challenge that is not well studied is *Change*. Facebook runs fast-evolving networks to adapt to varying application requirements. Changes occur not only in the data collection and processing stages but also when interpreted and consumed by applications. In this paper, we present PCAT, a production change-aware telemetry system that handles changes in fast-evolving networks. We propose to use a change cube abstraction to systematically track changes, and an intent-based layering design to confine and track changes. By sharing our experiences with PCAT, we bring a new aspect to the monitoring research area: improving the adaptivity and evolvability of network telemetry.

1 Introduction

Network telemetry is an integral component in modem, large-scale network management software suites. It provides visibility to fuel all other applications for operation and control. At Facebook, we built a telemetry system that has been the cornerstone for continuous monitoring of our production networks over a decade. It collects device-level data and events from hundreds of thousands of heterogeneous devices, millions of device interfaces, and billions of counters, covering IP and optical equipments in datacenter, backbone and edge networks. In addition to data retrieval, our telemetry system performs device-level and network-wide processing that generates time-series data streams and derives real-time states. The system serves a wide range of applications such as alerting, failure troubleshooting, configuration verification, traffic engineering, performance diagnosis, and asset tracking.

While our telemetry system can adopt algorithm and system proposals from the research community (e.g., [18, 27, 48, 50]), a remaining open challenge is *Change*. Changes happen frequently in our network hardware and software to meet the soaring application demands and traffic growth [16]. These changes have a significant impact on the network telemetry system. First, we have to collect data on increasingly heterogeneous devices. This is exaggerated as we introduce in-house built FBOSS [13], which allows switches to update as frequently as software. Second, we have growing applications (e.g., [1]) that rely on real-time, comprehensive, and accurate data from network telemetry systems. These applications introduce diverse and changing requirements for the telemetry system on the types of data they need, data collection

frequency, and the reliability and performance of collection methods.

The changes this paper considers include not only the network events from the monitored data, but also those updates to the telemetry system itself: modification to monitoring intent, advance of device APIs, adjustment of frequency configurations, mutation of processing, and restructure of storage formats. Without explicitly tracking them in our network telemetry system, we struggle to mitigate their impact to network reliability. For example, a switch vendor may change a packet counter format when it upgrades a switch version without notifying Facebook operators. This format change implicitly affects many counters in our telemetry database (e.g., aggregated packet counters), leading to adverse impact to downstream alerting systems and traffic engineering decisions. This example highlights several challenges: (1) Production telemetry is a complex system with many components (e.g., data collection, normalization, aggregation) from many teams (e.g., vendors, data processing team, database team, application teams). A change at one component can lead to many changes or even errors at other components. As a result, when telemetry data changes, it is difficult to discern legitimate data changes from semantic changes. (2) Sometimes, we only detect the error passively when traffic engineering team notices congestion. Yet, we cannot diagnose it easily because the error involves many data. Even worse, it may only affect a small portion of vendor devices due to phased updates. Section 2 shares more such examples.

In this paper, we propose to treat changes as first-class citizens by introducing PCAT, a Production *Change-Aware* Telemetry system. PCAT includes three key designs:

First, inspired by the database community [8], we introduce the *change cube* abstraction for telemetry to explicitly track the time, entities, property, and components for each change, and a set of primitives to explore changes systematically. Using change cubes and their primitives, we conduct the first comprehensive study on change characterization in a production telemetry system (Section 3). Our results uncover the magnitudes and the diversity of changes in production, which can be used for future telemetry and reliability research.

Second, we re-architect our telemetry system to be change-aware and evolvable. In the first version of our telemetry system, we have to modify configurations and code at many devices every time a vendor changes the counter semantics or collection methods, or an application changes monitoring intents. To constrain the impact of changes, i.e., the number of affected components, PCAT includes an intent-based lay-

ering design (Section 4) which separates monitoring intents from data collection and supports change cubes across layers. PCAT enables change attribution by allowing network engineers with rich network domain knowledge to define intents while having software engineers building distributed data collection infrastructure with high reliability and scalability. PCAT then compiles intents to vendor-agnostic intermediate representation (IR) data model, and subsequently to vendor-specific collection models, and job models. The intent-driven layering design reduces the number of cascading changes by 54%-100%, and enables systematically tracking changes through the monitoring process.

Third, we build several change-aware applications that explore the dependencies across change cubes to improve application efficiency and accuracy. For example, Toposyncer is our *topology derivation* service that builds on telemetry data and serves many other applications. We transformed Toposyncer to subscribe to change cubes based on derivation dependencies and greatly reduce topology derivation delay by up to 118s. We leverage correlation dependencies across change cubes to enable troubleshooting and validation.

The main contribution of this paper is to bring the community's attention to a new aspect of telemetry systems—how to adapt to changes from network devices, configurations, and applications. We also share our experiences of building change-aware telemetry systems and applications that can be useful to other fast-evolving systems.

2 Motivation

To keep up with new application requirements and traffic growth, data center networks are constantly evolving [16]. As a result, changes happen frequently across all the components in telemetry systems, ranging from device-level changes, collection configuration changes, to changes in the applications that consume telemetry data.

Our first generation of production telemetry system was not built to systematically track changes. This brings significant challenges for telemetry data collection at devices, integration of telemetry system components, debugging network incidents, and building efficient applications. In this section, we share our experiences of dealing with changes in our telemetry system and discuss the system design and operational challenges for tracking changes.

2.1 Bringing changes to first-class citizens

We motivate the needs of treating changes as first-class citizen in network telemetry with a few examples.
1. Build trustful telemetry data. Many management applications rely on telemetry data to make decisions. However, in production, telemetry data is always erroneous, incomplete, or inconsistent due to frequent changes of devices and configurations. Moreover, there are constant failures in large-scale networks (e.g., network connection issues, device overload, message loss, system instability). Therefore, applications need

to know which time range and data source are trustful and how to interpret and use the data. This requires tracking changes for each telemetry data value and semantics.

For example, we collect device counters at various scopes (e.g., interfaces, queues, linecards, devices, circuits, clusters). These counters may have different semantics with device hardware and software upgrades or network re-configurations. For example, we have a counter for 90th percentile CPU usage within a time window of a switch. When we change the switch architecture to mulitple subswitches [13], we set the counter as the average of 90th percentile CPU of subswitches. However, our alert on this counter cannot catch single subswitch CPU spikes that caused bursty packet drops. We need to know when to change the alerts based on counter changes.
2. Track API changes across telemetry components. Our telemetry system consists of multiple data processing components, which are independently developed by different vendors and teams. When one component changes its interfaces, many other components may get affected without notice. There are no principled ways to handle such changes across telemetry components. For example, vendor-proprietary monitoring interfaces often get changed without an explicit notification or detailed specification. This is because telemetry interfaces are traditionally viewed as secondary compared to other major features. However today cloud providers heavily rely on telemetry data for decisions in a fine-grained and continuous manner. If we do not update data processing logic based on device-level changes, the inconsistency may cause bugs and monitoring service exceptions.

In one incident, a routing controller had a problem of unbalanced traffic distribution, caused by incomplete input topology: a number of circuits were missing from the derived topology. This took the routing team and the topology team over three days to diagnose. The root cause was an earlier switch software upgrade that changed the linecard version from integer (e.g., 3) to string (e.g., 3.0.0). Such a simple format change was not compatible with the post-processing code that aggregated the linecard information into a topology. Thus, we missed several linecards in the topology, which then mislead TE decision and cause congestion in the network. This is not a one-off case, given many vendors and software versions coexist in our continuously evolving networks.
3. Debug with change-aware data correlation. As telemetry components keep evolving, it is hard to attribute a problem to a change using data correlation without explicitly tracking changes and their impacts. For example, when we fail to get a counter, the problem can come from data collection at the device, the network transfer, or both.

In production, we make changes in small phases: first canary on a few devices serving non-critical applications, then gradually on more devices to minimize disruptions to the network [13]. In one incident, there were a small number of devices with "empty data" errors for a power counter. The errors increased gradually and ultimately went beyond 1% threshold
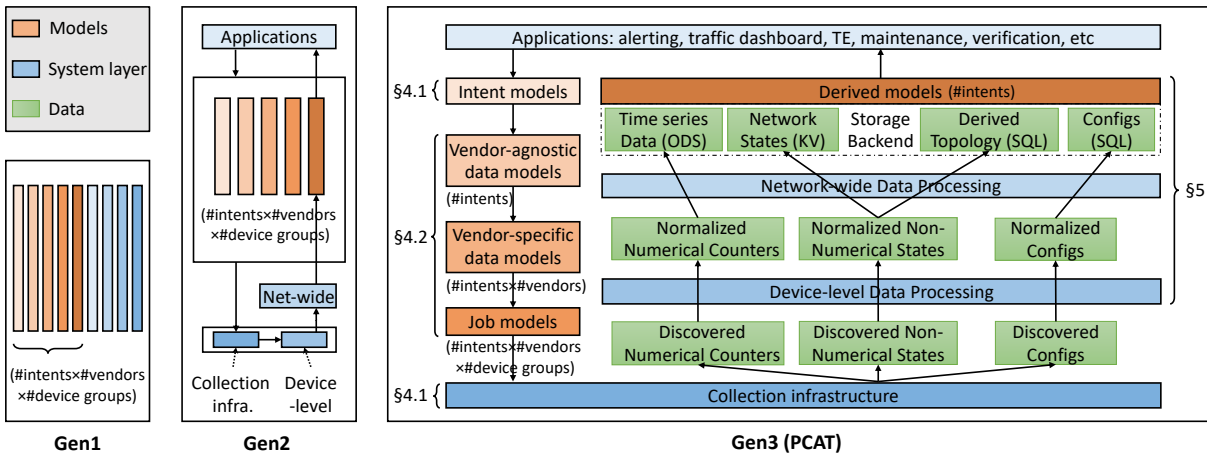
Figure 1: Generations towards change-aware telemetry.

after two weeks and triggered an alarm. This problem was difficult to troubleshoot due to its small percentage. We manually explored the changes through correlation: checking whether there were code changes before the failure, whether the failed devices shared a common region (indicating regional failure), a common vendor, or on common data types. We tried many dimensions of correlation and finally found the errors were mostly related to power and environment counters. The root cause was a vendor changing its format but the processing code could not recognize it. This example shows a tedious manual process of data correlation to debug problems because of gradual change rollouts. To improve debugging, we need to use changes to guide data correlation.

2.2 Lessons from Previous Generations

We now discuss our previous two generations of telemetry systems prior to PCAT and their limitations in handling changes.
Gen1: Monolithic collection script. In a nutshell, a telemetry system is a piece of code that collects data using APIs from the devices. Our first generation is naturally a giant script that codifies what counters to collect. It hardcodes the collection method, polling frequency, post-processing logic, and where to store the data. Figure 1 illustrates Gen1 as intertwined models and system blocks. It runs as multiple cron jobs, each collecting data from different groups of devices. This design is intuitive to implement but is not change-friendly. If a vendor changes the format of a counter, we need to sweep through the entire script to change the processing logic accordingly, and redeploy the new code to all monitors. It has high maintenance burdens as it relies on expert's deep understanding of the code to make changes. Further, tracking changes relies on version control system in the form of code differences, which do not reveal the intent directly.
Gen2: Decoupled counter definition from collection process. As our network expanded, the hulking script in Gen1 became hard to manage. We moved to Gen2, which separates the monitoring model (i.e., what counter to collect) from the actual collection code, shown as orange and blue boxes in Fig-

ure 1. The separation allows us to track changes to data types separately from the collection logic (e.g., sending requests, handling connections). However, the intent is still mingled with the vendor-specific counter definition. For instance, one may want to collect the "packet drops per interface". One needs to specify the exact SNMP MIB entry name and the specific API command. A low-level format change would result in updates on all model definitions. Moreover, the data collection system includes both the collection infrastructure and data processing logic. The data processing logics scatter across many places, e.g., when the data is collected locally at the collector, or before it is put into the storage. To change a piece of processing logic, we have to change many such places, which is cumbersome to track. In addition, when a piece of data is changed or is absent, tracing back on what causes the change is manual and tedious.

2.3 Challenges and PCAT Overview

Our experiences of previous two generations indicate three main challenges in handling changes: change abstraction, attribution, and exploration. To address these challenges, we build our Gen3 telemetry system – PCAT.
Change abstraction. In Gen1 and Gen2, changes were not stored structurally. They exist either as diffs in code reviews in Gen1, or logs to temporary files in Gen2. Without a uniform representation, each application needs to develop ad-hoc scripts to parse each data source. This leads to not only duplicate efforts but also missing changes or mis-interpretations. A uniform and generic change abstraction allows hundreds of engineers to publish and subscribe to changes to boost reliable collaboration without massive coordination overhead. In §3, we propose a generic abstraction called *change cube* to tackle this challenge.
Change attribution. The second challenge is the turmoil to ascribe the intent of the scattering changes. The solution involves a surgically architectural change to a multi-layer design, shown in Figure 1 and elaborated below.

Data collection. The first step is to collect data from de-

vices, called *discovered data*. There are three types: numeric counters, non-numeric states, and configurations (see Table 4 in Appendix). We use different protocols for collecting different data and for different devices: SNMP [10], XML, CLI, Syslog [28], and Thrift for our in-house switches [13].

Device-level data processing (normalization). The data is different in formats and semantics across devices, vendors, and switch software. This makes it difficult for applications to parse and aggregate the data from different devices. We use a device-level data processing layer to parse the raw data to a unified format across devices, vendors, and switch software.

Network-wide data processing. Next, we aggregate device-level (normalized) counters, states, and configurations into network-wide storage systems for applications to query. The normalized non-numerical states (as network states) are stored in a key-value store. We build a tool called **Toposyncer** which constructs *derived topology* from normalized non-numerical states. For example, from per-device data, we can construct the device, its chassis, linecard, as well as cross-device links.

Data consumer applications. There are many critical network applications that consume PCAT data. Network health monitoring and failure detection use monitoring data to detect and react to faults. Network control relies on real-time data for making routing and load balancing decisions [2, 38]. Maintenance and verification use telemetry data to compare network states before and after any network operations.

There are several advantages of the new design compared to previous generations. First, compared to Gen2, Gen3 dissects a monolithic data definition into three different types, each focusing on defining one aspect of the monitoring. The separation brings better scalability and manageability. We describe the details in §4. Second, we not only care about tracking changes in data format and code, but also need to attribute changes to the right teams (i.e., who/what authored the change). Change attribution builds the trust of the data for applications. It facilitates collaboration across teams towards transparent and verifiable system development. Gen3's intent-based layering design lets each team play by their strength and work together seamlessly. Specifically, the network engineers can leverage their rich domain knowledge and focus on intent definition, while software engineers focus on scaling the distributed collection system.

Change exploration. Many designs and operations require a clear understanding of the relations amongst changes. For example, to debug why a piece of data is missing, we always find the last time the data appears and check what has changed since then. We may find one change to be the cause, which could be caused by another change somewhere else. Similarly, when receiving a change of an interface state, we need to reflect the change on the derived topology and upper-layer applications. It motivates us to develop primitives for change exploration that serves many applications. We demonstrate the usage in real-time topology derivation in §5.

3 Changes in Facebook Network Telemetry

In this section, we define the change cube concept and explain how they are generated in this system, together with extensive measurement results by composing queries on top of the change cubes.

3.1 Change Cube Definition

To systematically handle changes in network telemetry, we leverage the concept of *change cubes*. Change cubes are used in databases [8] to tackle the data change exploration problem by efficiently identifying, quantifying, and summarizing changes in data values, aggregation, and schemas. Change cube defines a set of schemas for changes and provides a set of query primitives. However, changes in network telemetry are different from those in databases in two aspects: (1) Network telemetry generates streaming data with constant value changes, so the change cubes in network telemetry do not care about value changes but only changes in schema and data aggregation. (2) Network telemetry has frequent changes due to fast advances of hardware and software that result in data semantics changes.

Change cubes. We define a change cube to be a tuple $< Time, Entity, Property, Type, Dependency >$. We summarize each field of the change cube in Table 1 and explain below.

- *Time* dimension captures when the change happens. It depends on the granularity we detect changes, e.g., seconds, minutes, or days.
- *Entity* represents a measurement object, e.g. a switch, a linecard, as well as the models that describe what to measure and how.
- *Property* contains the fields or attributes of the entity that get changed. For example, a loopback IP address of a switch, an ingress packet drop of an interface.
- *Layer* dictates the layer or component in the telemetry system (in Figure 1) where changes happen. We discuss how we land in these choices in §4.
- *Dependency* dimension contains a list of other changes that this change is correlated with. Each item in the list is a $< ChangeCube, DependencyType >$ pair. We support two dependency types: correlation dependency and derivation dependency. Derivation dependency means that a lower-layer change causes an upper-layer change. Correlation dependency means two changes on correlated entities or properties.

Primitives on change cubes. Next, we introduce the operators on the change cube, which are used to explore the change sets. We leverage the operators proposed in [8] but redefine and expand them in the context of telemetry systems.

- $Sort_f(C)$ applies function f to a set of change cubes C, on one or a few dimensions to a comparable value, and uses it to generate an ordered list of C. In our problem, sort is mainly used with time to focus on the most recent changes.
- $Slice_p(C)$ means selecting a subset of C where the predicate

Dimension	Sub category	Examples
Time	Multiple time granularities	Second, minute, hour, day
Entity	Intent model	High-level intent, e.g., packet drops at spine switches
	Vendor-agnostic data model	Counter scope, unit
	Vendor-specific data model	Format, API
	Job model	Collection channel, frequency, protocol
	Derived model	Derived network switch
Property	Model fields	IP address, network type
	Change attributes	LoC, reason
Layer	Application	Adding alert to detect a new failure type
	Network-wide processing	Topology discovery code logic
	Device-level processing	Normalization rule
	Collection infrastructure	Codebase for collection tasks
Dependency	Correlation dependency	BGP session and interface status
	Derivation dependency	Circuit is derived from two interfaces' data

Table 1: Change Cube Definition.

p is true. It is used to filter an entity or a property value.

- $Split_a(C)$ partitions C to multiple subsets by attribute a. An example is to split the changes by the layer to group changes according to where they occur. A reverse operator to *Split* is *Union*, which combines multiple change sets.
- $Rank_f(P_C)$ After we split C to multiple sets P_C, we further analyze these sets and rank them based on a function, e.g., cube size, the time span, the volatility.
- $TraceUp(c)$ and $TraceDown(c)$. These two operators are used with the *Dependency* field, which are new compared to [8]. The former traces the changes that the current change c depends on, and the latter traces the changes that depend on the c. They are useful for debugging through layers and validation across data.

Explicitly tracking changes in a structured representation eases the diagnosis process. Considering the second example in §2.1, when the switch software is updated, it populates a change cube to the database, indicating the API's return result has changed. Consequently, it triggers another change cube at the counter model level on this specific CPU counter. This change cube in turn propagates through the monitoring stack to job changes and retrieved data changes. The applications using the CPU counters can subscribe to such data change, which can then be notified immediately. The chain of change notifications eliminates the post-mortem debugging after the counter change causes application errors.

3.2 Changes in PCAT

Leveraging *change cubes*, we provide the first systematic study of changes and their impact on network telemetry systems. We populate change cubes of PCAT using multiple ways. For the data stored in database, we leverage our database change pub/sub infrastructure [39]. We subscribe to the telemetry objects' change log and translate them to change cubes. For code changes in collection infrastructure, data processing logics (both device-level and network-wide), and

Queries	Formulas
Q1 (Fig. 2a)	$Sort_{Time(Week)}(Slice_{layer="application"}(C))$
Q2 (Fig. 2a)	$Sort_{Time(Week)}(Slice_{entity="vendor-agnostic\ data\ model"}(C))$
Q3 (Fig. 2c)	$\sum_c c.LoC, c \in Split_{Time(Week)}(Slice_{layer="application"}(C))$
Q4 (Fig. 3a)	$Sort_{Time(Day)}(Slice_{entity="job\ model"\ \&\ property="frequency"}(C))$
Q5 (Fig. 3b)	$Split_{network\ type}(Slice_{entity="job\ model"\ \&\ property="frequency"}(C))$
Q6 (Fig. 3c)	$Split_{network\ type}(Slice_{entity="job\ model"\ \&\ property="channel"}(C))$
Q7 (Fig. 4a)	$Split_{blueprint\ type}(Slice_{layer="application"\ \&\ reason="blueprint"}(C))$
Q8 (Fig. 4b)	$Split_{vender}(Slice_{layer="application"\ \&\ reason="new\ model"}(C))$

Table 2: Queries used in §3.2.

applications, we parse the logs in the code version control system to generate change cubes. Intent model, data model (both vendor-agnostic and -specific), and job model changes are codified and thus tracked through code changes [41]. They can be populated using the same way as other code changes. We store all change cubes to a separate database called *ChangeDB* and develop APIs to explore these changes.

We analyze changes from the perspectives of devices, collection configurations, and application intents, over seven years (2012-2019). Our results below uncover surprisingly frequent changes and quantify the diverse causes of changes.

3.2.1 Change Overview

Change frequency. We first quantify the code changes of our monitoring system. We map one code commit to one change cube, involving multiple lines of code across multiple files. We group the changes into three categories according to where they happen in Figure 1: collection infrastructure (bottom layer), data & job models and processing (middle two layers), and applications, representing the infrastructure, data, and intent respectively. We construct queries using the primitives defined earlier. We put the actual query to generate the figures in Table 2. Q1 uses *Slice* to filter the changes in application layer, and sorts the changes by time. We replace the "application" with other values for changes in other layers. Q1
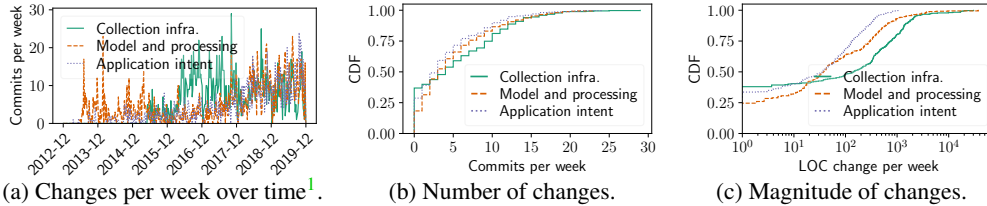
(a) Changes per week over time[1].

(b) Number of changes.

(c) Magnitude of changes.

Change reasons	%
Collection infrastructure	67.9
Adding new devices	17.8
Topology processing	8.30
Data format	4.86
Counter processing	1.19

Table 3: Change categorization by change reasons.

Figure 2: Change characteristics.
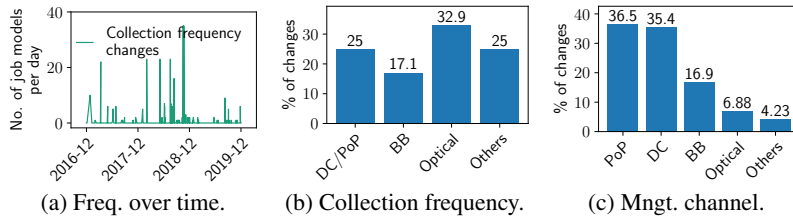
(a) Freq. over time.

(b) Collection frequency.

(c) Mngt. channel.

(a) Add new data type.

(b) Blueprint.

Figure 3: Collection configuration changes[2].

Figure 4: Network configuration changes.

can be compiled into the following SQL: `SELECT COUNT(*) FROM ChangeDB WHERE layer = "application" GROUP BY time_week ORDER BY time_week`.

Figure 2a shows the number of changes per week. We find that *types of changes vary greatly as the telemetry system scales*. More model and processing changes occur at the beginning (the year of 2013), as we begin by adding more counters to monitor. Wh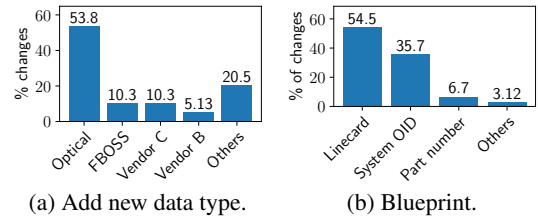en the number of counters reaches a certain scale (the year of 2016), we realize the infrastructure needs better scalability. Thus there are more changes to refactor the collection infrastructure. Application intent follows the same trend as data changes, as adding new data is often driven by the need from applications.

Cumulatively across time, we show the average numbers of weekly changes of three categories in Figure 2b. They are on the same order of magnitude, with slightly more infrastructure changes. It can be as high as 25-30 changes per week. Note that each change is deployed on many switches and the changes it introduces to the network is significant.

Change magnitude. We quantify the magnitude of changes in terms of Lines of Code (LoC) using query Q3. While most code changes are not big, some changes could touch multiple lines due to consolidation of processing logic and refactoring. This is obtained by first getting a slice of changes of a given category, splitting the changes into weeks, and summing up the LoC property. Figure 2c shows that *collection infrastructure has larger changes and the application has changes with larger volatility*. We can dig into the volatility of each change set by computing its variance and use the *Rank* primitive. Both figures show there exists a significant number of large changes. For example, there are 27 weeks with more than 1000 LoC changes for collection infrastructure. However, as the industry's trend is to move away from monolithic changes

---

[1]The collection and processing infrastructure were not merged into the codebase before 2015-04; so its commits are non-trackable before that.

[2]The "Other" contains some changes that are hard to classify programmatically. The same applies to Figure 4.

---

to many small incremental changes [13], we expect to have more frequent small changes going forward.

Change reason categorization. We analyze the breakdown by reason of change using $Split_{reason}(C)$, which is obtained by parsing the commit log text and adding it to the ChangeDB. Table 3 shows that one major reason is collection infrastructure changes (67.9%). Adding new devices to the network is the second dominant reason (17.8%). Topology processing changes occupy 8.3%. The fourth reason is adjusting the data formats of collection models (4.86%). Lastly, 1.19% come from the device-level counter processing code.

3.2.2 Device-Level Changes

In our large-scale networks, we constantly add new vendors and devices to leverage a rich set of functions and to minimize the risk of single-vendor failures. The number of devices increased 19.0 times and the number of vendors increased 4.7 times as observed by PCAT in six years. Even with the same vendor, we gradually increase the chassis types, which have different combinations of linecard slots and port speeds. More choices of chassis types allow us to have fine-grained customization to our network needs. Furthermore, the number of chassis types grows from 26 to 129 (4.4×). In addition, our in-house software switch has tens of code changes daily and deploys once every few days [13].

3.2.3 Collection Configuration Changes

Collection frequency. Applications adjust the collection frequency to balance between data freshness and collection overhead. We first analyze collection changes by counting daily changes of collection frequencies over time, using query Q4. Figure 3a shows that there are constant collection frequency changes over time, with more frequent changes near December 2018 – because of tuning collection frequencies for newly-added optical devices. We analyze collection frequency changes by applying *Slice* on both the entity and the property. Interestingly, Figure 3b shows that *optical devices*

change frequency more often (32.9%) because they cannot sustain high-frequency data polling and thus require more careful frequency tuning.

Management channel changes. PCAT collects data from the management interfaces at devices. As our management network evolves, we frequently reconfigure management interfaces (e.g., IP addresses, in-band vs. out-of-band interfaces). Backbone and PoP devices have multiple out-of-band network choices for high failure resiliency. Figure 3c breaks the IP preference changes into PoPs, DCs, and Backbones. *PoPs have more frequent channel changes (36.5%)* because PoPs are in remote locations and thus have more variant network conditions. Selecting the right channel is important to keep the device reachable during network outages so that we can mitigate the impact quickly.

3.2.4 Application Intent Changes

Data type changes. PCAT supports an increasing number of diverse applications over years. Applications may add new types of data to collect (e.g., to debug new types of failures), or remove some outdated data. Figure 4a shows how different vendors add new data types. Optical device vendors add more data (53.8%) because we recently start building our own optical management software and thus need more counter types. Indeed, optical devices generally have more types of low-level telemetry data compared to IP devices, e.g., power levels, signal-to-noise ratio. They are also less uniformed across vendors than IP devices.

Hardware blueprint changes. Hardware blueprint specifies the internal components (chassis, linecards) of each switch and determines what data to collect. Figure 4b shows the percentage of changes for hardware blueprints such as linecard map, system Object Identifier (OID) map, part number map, and others (e.g., OS regex map). These changes are due to network operations such as device retrofit and migration. They may cause data misinterpretation if not treated carefully.

4 Change Tracking in Telemetry System

In this section, we describe the layering design of our current intent-based telemetry system to help track changes.

4.1 Towards change-aware telemetry

Intent modeling. We use a thrift-based modeling language that empowers network engineers to easily specify their monitoring intents. Compared to other intent language proposed in academia [19, 31, 32], our language puts more emphasis on device state in addition to traffic flows, and defines actions in addition to monitoring. Our language contains three components shown in Figure 5.

- *Scope* captures both the device-level scope (e.g., Backbone Router) and network-level scope, (e.g., DC fabric network).
- *Monitor* specifies what to monitor in a vendor-agnostic way. For example, an intent could be capturing packet discard for the gold-level traffic class, which will get translated
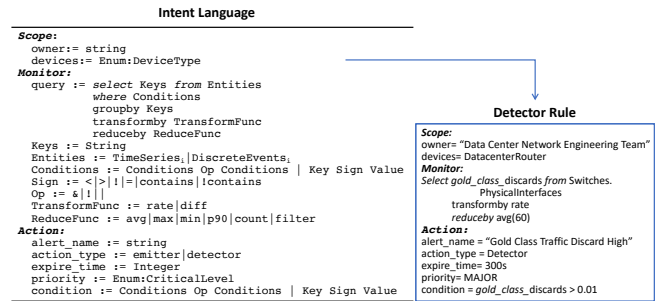
Figure 5: Intent model.
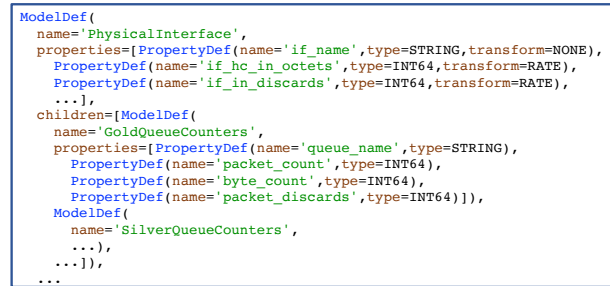
Figure 6: Data model.

to a specific SNMP MIB entry or particular counters. In the left part of Figure 5, we describe the SQL-like query language. The *keys* are monitored metrics and the *entities* are time-series data streams and discrete events tables. We also support data aggregation functions such as *avg*, *count*, *filter*, which aggregate samples over time and devices.

- *Action* includes two types: Emitter and Detector. Emitters subscribe to *discrete network events* that are pushed from devices, and define actions upon receiving these events. Detectors allow us to write formulas for various time-series data, and set up a threshold for the formula value as the alerting condition. A detector example is shown in the right part of Figure 5; it defines a detector based on the key *gold_class_discards* which captures the packet drops for gold-class traffic on a physical interface. The discard is transformed to rate, and aggregated every 60 seconds. The alert is triggered if the threshold is greater than 0.01.

The intent model hides low-level changes. Vendors may change the queue drop counter names, or the mapping between queues and gold-class traffic may change. The intent configuration remains unchanged in both cases.

Runtime system. We handle heterogeneous intents with homogeneous software infrastructure. Thanks to separation, software engineers can focus on the runtime execution system to solve the hard system building problems: scheduling, load balancing, scalability, and reliability. The runtime execution system collects data from devices according to the model, which includes a distributed set of engines and a centralized controller to distribute jobs and collect data from these engines. The centralized controller fetches the latest collection and job models, combines with device information in our database, and generates a sequence of jobs to be executed

```
ModelDef(
   ...
   name='GoldQueueCounters',
   ...
).vendor1_implementation(
   name='Vendor1_Cli_Impl',
   engine=CLI,
   engine_options=EngineOptions(
      command='show interfaces {$if_name} red drops',
      parser='parse_cli_vendor1_phy_if')
).vendor2_implementation(
   name='Vendor2_Thrift_Impl',
   engine=Thrift,
   engine_options=EngineOptions(
      function='getQueueDrops($high_pri_queues)',
      parser='parse_thrift_vendor2_phy_if')
   )
).vendor3_implementation(
   name='Vendor3_Cli_Impl',
   engine=CLI,
   engine_options=EngineOptions(
      command='show interfaces discards {$high_pri_queues}',
      parser='parse_cli_vendor3_phy_if')
   )
)
```

```
Job(
   model_tree='GoldQueueCounters ',
   backend_settings=[ODS('15m')],
   device_impls=[
      DeviceImpl(include_filter=DeviceFilter(
         os_types=[OSType.VENDOR1_OS],
         device_roles=['Rack Switch']),
         implementation= 'Vendor1_Cli_Impl',
         parser='parse_cli_vendor1_phy_if'),
      DeviceImpl(include_filter=DeviceFilter(
         os_types=[OSType.VENDOR2_OS],
         device_roles=['Fabric Switch']),
         implementation= 'Vendor2_Thrift_Impl',
         parser='parse_thrift_vendor2_phy_if'),
      ...
   ])
```

```
# show interfaces eth1 red drops
......
  MTU 1500 bytes, BW 100000 Kbit/sec, DLY 100 usec,
......
  1 minute output rate 5M bits/sec, 5304 packets/sec
......
  245 output errors, 568 pkt drops, 2 interface resets
......
```

```
def _parse_Cli_Vendor1_Interface_Resets (ot, data):
   otn=ot.root.add_child(ot.type.root)
   for line in data:
      regex_match=re.match('.*\s(\d+)\s+ pkt drops', line)
      if regex_match:
         ......
      return [otn]
```
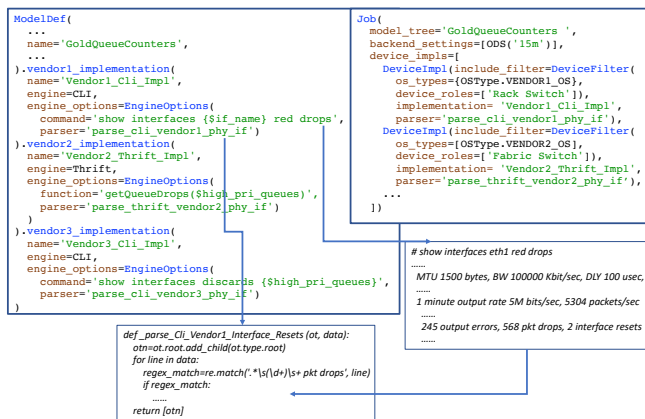
Figure 7: Collection method and job model.

with a given deadline. It dispatches jobs to designated engines based on load and latency. The engine executes the collection command, performs device-level processing, and sends data back to the corresponding storage. The system is heavily engineered to tackle the reliability and scalability challenges.

4.2 Change reduction w/ vendor-agnostic IR

Next, we zoom into the intermediate layer between the intent and the runtime. The high-level monitoring intent is translated to the intermediate representation data model, which gets mapped to the vendor-dependent collection model, and finally is materialized to the job model on each device. We emphasize that how the modular design principle is translated to different models in order to *limit the impact of changes*.

Vendor-agnostic intermediate representation (IR) data model. The data model is created based on the *keys* field in the intent model. It specifies data schema in the following way, as shown in Figure 6.

- *Hierarchical.* We choose a tree structure as an intermediate representation, called the *model tree*. An example is shown in Figure 7. An AggregateInterface model has multiple child models, e.g., PhysicalInterface, BGPSessions. A PhysicalInterface also has multiple child models. Models are like templates waiting to be filled in. When they are materialized by actual monitoring data, we call them **objects**. By organizing the materialized objects in the same hierarchy as the model tree and adding a dummy root to connect up the top-level objects, we get a materialized **object tree**. The models define the data to be collected, which is derived from the keys field in the intent model.
- *Typed.* The data model defines the types of data to make interpretation of the data easier, e.g., if_hc_in_octets is the incoming traffic in octets.
- *Processing instruction.* It also defines basic processing primitives to go with the data using the *transform* field, e.g., computing a per-second rate from consecutive absolute counts. Both the type and the processing instruction are determined by the intent. Placing all the processing logic in a separated blob makes it much easier to track the changes

in processing logic.

Vendor-dependent collection model. The IR model is further compiled down to vendor-specific counter names, specific commands to use, e.g., a CLI command, Thrift function name. Figure 7 shows two collection methods for the GoldQueueCounters data model: CLI and thrift. In each implementation, we define the collection API and the post-processing function in the *parser* field. We show an example of the CLI parser function that matches the regex in the output of a command on the vendor1 device. Creating this layer of model separately allows us a place to capture all changes due to vendor format and API changes, which are quite common.

Vendor-dependent job model. The job model combines the collection method with a concrete set of devices, shown in Figure 7. The *implementation* field matches with what is defined in the collection method. Instead of defining a job spec for each device, we group devices and apply the same job spec for all of them. Figure 7 uses *DeviceFilter* to define device role (e.g., rack switches), OS type, region, device state, etc. Job models are the input to the runtime execution to handle job scheduling and manage job completion. Job model captures the system aspects of changes. It can be adapted according to performance and scalability requirement, which is independently controlled from the intent or data specifications.

5 Change Exploration

Once PCAT collects data based on monitoring intents, we run device-level and network-level processing to report the data back to applications. Below, we build a few change-aware applications by exploring dependencies across change cubes.

5.1 Change-driven Topology Derivation

Toposyncer is our topology generation service, part of the collection infrastructure (see network-wide data processing in Figure 1). It creates *derived topology* from normalized device-level data (i.e., in vendor-agnostic format) (Figure 8). For example, from per-device data (e.g., interface counters, BGP sessions), Toposyncer constructs the device, its chassis, linecard, as well as cross-device links.

Toposyncer overview Toposyncer has four processes: *(1) Sync_device* constructs nodes with multiple sub-components: sub-switches, chassis, line cards (line 2-11 in Figure 9). It also derives device-level attributes such as power and temperature, control and management plane settings. *(2) Sync_port* derives physical and logical interfaces on each node and their settings (IP address, speed, QoS) (line 12-13). *(3) Sync_circuit* constructs cross-device circuits. A circuit is modeled as an entity with two endpoints, pointing to the interface of each end's router [41]. For each interface, it searches for all possible neighbors based on various protocol data, e.g., LLDP, MAC table, LACP table. In case some data source is incomplete, we search all data sources, independently identify all possible neighbors from each data source, and consolidate the results.
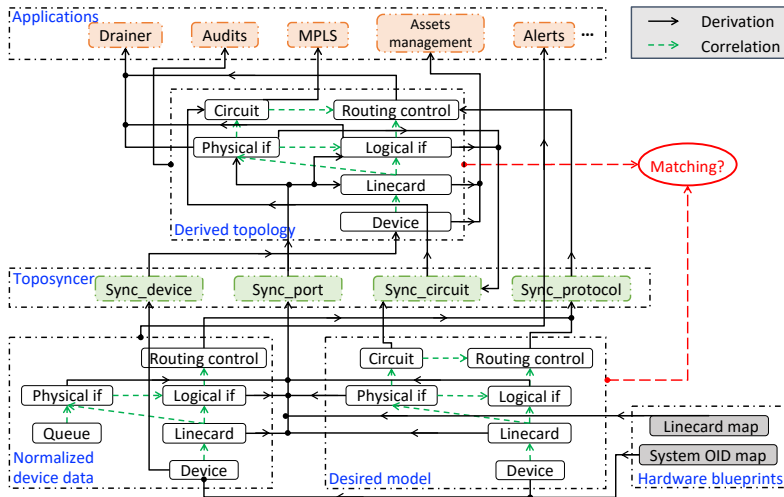
Figure 8: Data dependency graph. Toposyncer consumes the device data, desired model and hardware blueprints to generate derived data. PCAT verifies the derived data with the desired model.

```
 1: procedure DERIVETOPOLOGY(Collection, Desired,
       HdwTemps, DependencyG)
 2:     for d ∈ Desired.Devices do
 3:         DeviceObj, dep = sync_device(Collection, d)
 4:         DependencyG.add(dep)
 5:         blueprint = getBlueprint(HdwTemps, d)
 6:         for chassis_temp ∈ blueprint do
 7:             derived_chassis = sync_chassis(Collection,
                   chassis_temp)
 8:             for linecard_model ∈chassis_temp do
 9:                 derived_chassis.add(sync_linecard(Collection,
                       linecard_model))
10:             DeviceObj.addchassis(derived_chassis)
11:         DeviceObjs.add(DeviceObj)
12:     for d ∈ DeviceObjs do
13:         derived_ifaces = sync_port(Collection, d)
14:         for iface ∈ derived_ifaces do
15:             neighbors.add(findNeighbors(iface, Collection))
16:         circuits = sync_circuits(iface, neighbors)
17:         sync_protocol(Collection, circuits)
18: procedure UPDATETOPOLOGY(UpdateQ, DependencyG)
19:     while UpdateQ ≠ ∅ do
20:         change_i = UpdateQ.pop()
21:         dependent_objs = change_i.Dependency
22:         update_func=findFunc(dependent_objs)
23:         update_func(dependent_objs, change_i)
```

Figure 9: Toposyncer algorithm

(4) Sync_protocol creates the protocol layers on top of the circuits, such as OSPF areas, BGP sessions and their states.

Toposyncer uses two additional data sources as templates to guide the construction: the desired model which defines the operator's intent topology and hardware blueprints which include hardware specifications, as shown at the bottom in Figure 8. Figure 9 shows the process. It uses desired device data (names, IP addresses) to decide what device to derive (line 2). Then, it uses the hardware blueprints and desired data to handle ambiguity. For instance, to figure out "what is this chassis", it first checks the discovered chassis name in raw data from the device. But often the discovered name is not uniquely mapped to a chassis but to several possible chassis versions, e.g., two versions with 4 linecards, one version with 8 linecards. Toposyncer cross-checks with hardware blueprints and picks the best match[3] (line 6-8). This process is similar to other topology services [29, 41], but we focus on derived models and how we populate them automatically from telemetry data.

Improve Toposyncer with change cubes. Our first implementation of Toposyncer did not utilize changes. It ran periodically against the latest snapshots of collected data at a fixed frequency (e.g., 15 minutes). This method leads to stale derived data, which affects the freshness and accuracy of upper-layer applications. Another challenge is debugging. When a piece of data (e.g., a circuit) is missing in the derived topology, it is hard to find out whether it is because of a raw data change, a normalized data change, a desired model change, a hardware blueprint change, or other reasons. We tackle these problems using change cubes and the dependency primitives below.

Build change cubes. We generate change cubes for normalized data, desired model, hardware blueprint, as well as Toposyncer code changes, shown as each dotted box in Figure 8. We generate these cubes by parsing database transaction logs and model/code changes from version control system logs and publish them to ChangeDB. For example, when an operator changes the configuration of an SNMP MIB for a device, we generate a record to the DB.

Derivation dependency. We populate the derivation dependency across change cubes A and B if we derive data A from data B. In the above example, the MIB change will result in multiple change cubes of job models. We build the dependency between the MIB config change and the rest of job model changes. Figure 8 shows derivation dependency in solid arrows across objects in different layers (each large dash box representing a layer). The dependency exists between data objects as well as between code and data.

Subscription to change cubes. Toposyncer subscribes to the change cubes and invokes corresponding processing logic accordingly, shown in line 19-23. For example, *sync_port* subscribes to the device data (e.g., Thrift_Fboss_Linecards, Snmp_entPhysicalTable), and a hardware blueprint (i.e., linecard map). If the hardware blueprint changes, i.e., the same linecard name is mapped to a different hardware blueprint, the change cube will be published and *sync_port* triggers its function *sync_phy_iface* function on the impacted interfaces. Similar pub/sub relation is also built between applications and derived data. For instance, as shown in Figure 8, a drainer application subscribes to interface status and the routing control messages to determine if it is safe to perform an interface drain operation.

---

[3]When the guess is wrong, it exhibits as a discrepancy between desired and derived topology. We add alarming to detect such differences and involve humans to manually investigate.

5.2 Improve Trust on Data Quality

Real-world telemetry data may contain dirty or missing data. By exploring the change history, one can better judge whether the current values are trustworthy. Observing patterns of data changes can help predict the occurrences of future changes and identify missing changes.

Correlation dependency. Amongst normalized device data or derived data, data have relationships between them, shown as dash arrows within each large dash box in Figure 8. The relationship represents the physical dependency across objects, such as "contain", "connect", "originate". Previous topology-modeling works Robotron [41] and MALT [29] focus on the desired model and the correlation dependency in it. The desired model is built for the purpose of capturing topology intents and generating configurations. Here we use the model together with change cubes to verify if the actual topology's change is legit by comparing it against the desired models. A change cube generated from the desired object should have a matching change cube in the derived object, and vice versa. This can be done with a *Slice* on the entity, *Sort* by time, and compare the *Entity* of the changes.

This correlation dependency can also be used for cross-layer validation of data quality. We implement *if-then* validation rules based on the correlation dependency on change cubes. We give two examples below. One use case is hard-stop fault detection. One rule is that *if* the logical interface fails (i.e., a specific change cube on a logical interface), *then* the routing session going through it will also fail (i.e., another change cube on the routing session must exist). If we observe significant errors at the lower layer but no upper failure, it indicates a measurement issue. In another use case, the aggregate interface consists of multiple physical interfaces. *If* a member physical interface reports packet errors, *then* the packet errors from aggregated interface should be larger than or equal to the physical interface errors. If the rule is not satisfied, it indicates some issues. These cross-layer dependencies can help us detect change-induced problems more quickly.

6 Evaluation

This section evaluates how the layer design of PCAT has helped with change tracking and how much benefit the change cube method has brought to use cases.

6.1 Change tracking implementation

First, we examine whether tracking all the changes is even feasible in a production environment. We show the change cube data volume grows with time in Figure 10a. Drawing from the experiences of Facebook's data infrastructure team, we employ a two-tier storage solution. We have an in-memory database to hold the change data for the most recent 30 days and have a disk-based SQL database for longer historical data. At the same time, the change data is published to our publish/subscribe system [4] for real-time propagation. Next, we
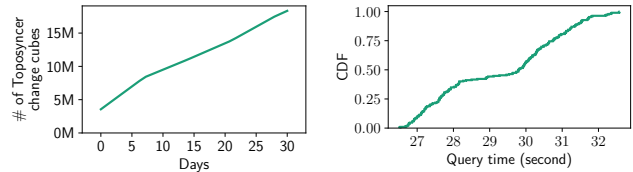
(a) # of change cubes over time. (b) Query distribution time.

Figure 10: Scalability and performance of change tracking.
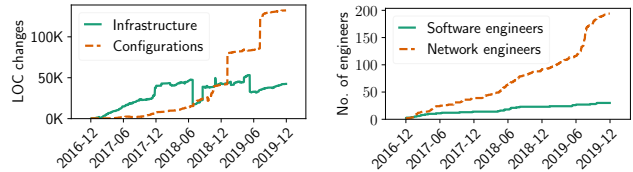
(a) LoC changes over time. (b) Engineers over time.

Figure 11: Separating configurations with telemetry infra.

evaluate the performance of exploration using the primitives defined in §3.1, which is implemented using SQL statements. Figure 10b shows the query distribution time for data stored on disk, most of which centers around 27-32 seconds, due to the large data volume. For shorter duration of data in memory, it takes less than one second.

6.2 Benefits of separation

Analyzing change data over time helps us evaluate the long-term benefit of the layer design. We show it from three aspects.

Decoupled evolvement of configurations and infrastructure. We categorize changes broadly to configuration changes vs. infrastructure changes. We quantify the magnitude of the change using the Lines of Code (LOC) change. Figure 11a shows that the changes for configurations are 3.1 times more than core collection infrastructure changes. The sudden jumps for configurations in January 2019 are due to adding a large set of optical devices, which was not monitored by PCAT. The second increase around July 2019 is due to the migration to Gen3, resulting in a large number of new models added. The result shows that we increase the monitoring scope by configuration layer changes with a stable infrastructure.

Scaling with divided responsibility. The separation in software systems has a long-term impact on the organization growth and people aspects. In Figure 11b, we analyze the change authors and categorize by their roles. It shows the number of network engineers who have made changes to configurations is increasing at a much faster pace than software engineers, with 7.2 times more people recently. The increase around June 2019 is due to both migration to Gen3 and adding more optical devices to monitor. It is clear that both of these changes are carried out by network engineers. It shows that PCAT enables network engineers to work on different network types while a small number of software engineers maintain infrastructure. It will boost a healthy collaboration environment where each team can play by their strength.

Confining the impact of changes. We use the number of change cubes as an approximate of the volume of changes.
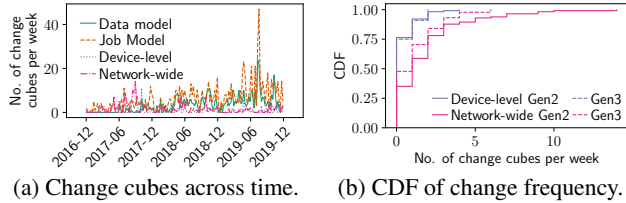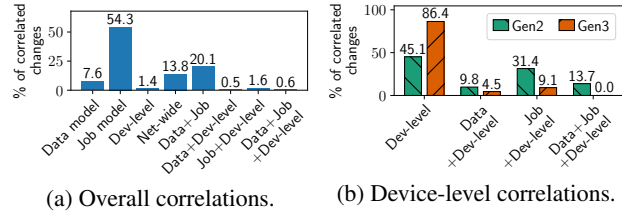
(a) Change cubes across time.

(b) CDF of change frequency.

Figure 12: Change cube frequency.

(a) Overall correlations.

(b) Device-level correlations.

Figure 13: Correlated changes.

(a) No. of changes to Toposyncer.

(b) Lagging reduction.

Figure 14: Change-driven Toposyncer.

Figure 12a shows its trends across time for data models, job models, device-level processing, and network-wide processing. The maximum numbers range from 15 to 50 for different categories. The models (data and job) have more changes due to the frequent intent changes. The infrastructure layers (device and network-wide processing) are more stable. Recently there are more data model and job model changes, because of Gen2-to-Gen3 migration. To directly illustrate the benefit of modular design in Gen3 (§4.2), Figure 12b compares the frequency of change cubes for device-level and network-wide processing in Gen2 and Gen3 (after 2019-02). We observe that the average change frequency for network-wide processing in Gen3 is 38.1% lower than Gen2, while device-level remains similar. This means the modular design in Gen3 further prevents the changes in lower data model and job model layers from impacting upper processing layers, confining the impact of lower-layer changes. Note that we discounted the changes due to Gen2-to-Gen3 migration to have a fair comparison.

Reducing correlated changes. We find change cubes that occur close in time as correlated changes (e.g., data and job models are modified in the same commit). We show that PCAT's way of separating layers and models has helped reduce correlated changes. We first present the breakdown of different correlation combinations in both generations in Figure 13a. The largest combination is data and job, accounting for 20.1%. It is because adding new devices requires adding both data and job models. There are a small fraction of changes that require updating data, job, and device-level processing all together. Most of them are due to adding some specific counters that require special processing. Figure 13b further breaks down all correlated changes related to device-level processing for Gen2 and Gen3. It shows that Gen3 has significantly reduced the correlated changes by 54.1%, 71.0%, and 100.0% (i.e., the second-to-last bar pairs) accordingly.

6.3 Benefits of change-driven Toposyncer

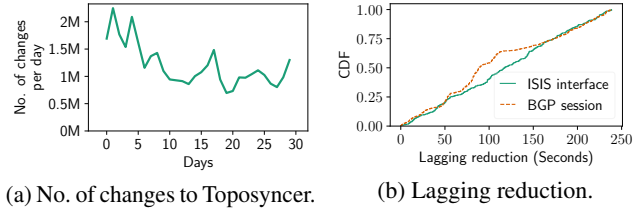The first benefit is explicitly tracking changes in a centralized manner. Figure 14a shows the magnitude of the changes over time to Toposyncer. Note that this is much higher than the changes presented earlier, since it includes the changes of raw data for network states.

The second benefit lies in the efficiency and accuracy improvement to applications. We evaluate it using the lagging time, i.e. the time between the change happening and when changes are reflected in derived topology by Toposyncer. Figure 14b shows the topology derivation is much more timely: reducing 118.76s lagging time for ISIS interface updates, and 108.93s for BGP session state updates, averagely.

7 Lessons and Future Directions

We discuss our lessons from building PCAT and the opportunities for future research.

Efficiency vs. adaptivity. We work closely with vendors to improve the efficiency of data collection primitives at switches (similar to academic work on reducing memory usage and collection overhead with high accuracy [24,26,46]). However, pursuing efficiency brings us challenges on adaptivity. Different devices have different programming capabilities and resource constraints to adopt efficient algorithms. Introducing new primitives also adds diversity and dynamics to upper layers in the telemetry system. For example, we work with vendors to support a sophisticated micro-burst detection on hardware. However, if only a subset of switches supports this new feature, applications need complex logic to handle detected and missed micro-bursts. Thus we have a higher bar for adopting efficient algorithms due to adaptivity concerns.

To support diverse data collection algorithms, we need a full-stack solution with universal collection interfaces at switches and change-aware data processing and aggregation algorithms. Recent efforts on standardizing switch interfaces such as OpenConfig [3] is a great first step but does not put enough emphasis on standardizing telemetry interfaces. Recent trends on open-box switches (e.g., FBOSS [13]) bring new opportunities to develop adaptive telemetry primitives.

Trustful network telemetry. Telemetry becomes the foundation for many network management applications. Thus we need to know which data at which time period is trustful. However, building a trustful telemetry system is challenging in an evolving environment with many changes of devices, network configurations, and monitoring intents. Fast evolution also introduces more misconfigurations and software bugs. Explicitly tracking change cubes and exploring their dependencies in PCAT is only the first step.

We need more principled approaches for *telemetry verification and validation* across monitoring intents, data models, and collection jobs. Compared with configuration verification work [7, 35], telemetry verification requires quantifying the impact of changes to the measurement results. One opportunity is that we can leverage cross-validations across multiple counters covering the same or related network states or across aggregated statistics over time. For example, we collect power utilizations (watts) from both switches and power distribution units (PDUs). In this way, we can validate the correctness of these utilization counters by comparing the PDU value with the sum of switch values.

Telemetry systems are complex time-series databases. We can leverage provenance techniques [12] to support change tracking, data integration, and troubleshooting. One challenge is that we cannot build a full provenance system due to vendor-proprietary code and network domain-specific data aggregation algorithms. There are also unexpected correlation dependencies across data.

Integration between telemetry and management applications. Our production networks are moving towards self-driving network management with a full measure-control loop. PCAT shows that changes bring a new complexity to the measure-control loop. Control decisions not only affect the network state that telemetry system captures but also the telemetry system itself. For example, an interface change may affect a counter scope. A traffic engineering control change may affect data aggregation because traffic traverses through different switches. These telemetry data changes in turn affect control decisions. We need to identify solutions that can feed control-induced changes directly into the telemetry systems.

Another question is how to present large-scale multi-layer telemetry data to control applications. Rather than providing a unified data stream, control applications can benefit from deciding what time, at what granularity, frequency, and availability level for data collection and the resulting overhead and accuracy in the telemetry system. One lesson we learned is to have the telemetry data available when it is mostly needed. For example, the network's aggregated egress traffic counter, which is collected at the edge PoPs, is a strong indicator of the business healthiness. To ensure its high availability, we need to give control applications the option to transfer the counter on more expensive out-of-band overlay networks. Moreover, we may extend the intent model to explicitly express the reliability-cost tradeoffs and adapt the tradeoffs during changes. We also need new algorithms and systems that can automatically integrate data at different granularities, frequencies, and device scopes to feed in control applications.

8   Related Work

Network evolvement. Several existing works have also pointed out the importance of considering changes. Both Robotron [41] and MALT [29] discuss it in the context of topology modeling, but miss the practical challenges of net-

work monitoring. [16] discusses network availability during changes, while we focus on telemetry systems during changes. **Other monitoring techniques.** PCAT is a passive approach. Active measurement injects packets into the network [14, 17, 18, 34, 49], and they are complements to passive measurement. The design principle of PCAT to handle changes can be applied to existing monitoring systems [20, 25, 36, 44, 48, 50], languages and compilers [9, 19, 32, 33]. PCAT also benefits from recent software-defined measurement frameworks [25, 27, 32, 46, 48]. For example, similar to OpenSketch [48], PCAT frees network engineers from configuring different measurement tasks manually. PCAT 's intent model design borrows ideas from the query language in Marple [32]. There are many memory-efficient monitoring algorithms [22, 23, 26, 30, 46] that focus on the expressiveness and performance of network monitoring. They provide adaptivity but only to a limited type of new queries, resource changes, or network condition changes. Here, PCAT focuses on a broader set of adaptivity (e.g., adaptive to counter semantics changes, data format changes, and more).

Dependency in network management. Dependency graph has been widely used for root cause localization [5, 6, 37, 42, 43, 47, 50]. Statesman [40] captures domain-specific dependencies among network states. We share some similarities but use dependency to tackle the change propagation.

Techniques from database and software engineering. Data provenance [12, 15] encodes causal relations between data and tables in metadata. Several works [11, 45] apply provenance to network diagnosis. [8] proposes the change cube concept and applies it to real-world datasets. All the above works focus on data face-value. On the other hand, software engineering community studies the problem of how a change in one source code propagates to impact other code [21, 51]. Ours looks at changes from telemetry systems from both data, configurations, and code.

9   Conclusion

This paper presents the practical challenge of a monitoring system to support an evolving network in Facebook. We propose explicitly tracking changes with change cubes and exploring changes with a set of primitives. We present extensive measurements to illustrate its prevalence and complexity in production, then share experiences in building a change-aware telemetry system. We hope to inspire more research on adaptive algorithms and evolvable systems in telemetry.

Acknowledgments

References

[1] Express backbone. https://engineering.fb.com/data-center-engineering/building-express-backbone-facebook-s-new-long-haul-network/.

[2] Introducing proxygen facebook c++ http framework. https://code.fb.com/production-engineering/introducing-proxygen-facebook-s-c-http-framework.

[3] OpenConfig YANG model. http://www.openconfig.net/projects/models/.

[4] Scribe. https://github.com/facebookarchive/scribe.

[5] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Boon Thau Loo, and Geoff Outhred. 007: Democratically finding the cause of packet drops. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 419–435, 2018.

[6] Paramvir Bahl, Ranveer Chandra, Albert Greenberg, Srikanth Kandula, David A Maltz, and Ming Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. *ACM SIGCOMM Computer Communication Review*, 37(4):13–24, 2007.

[7] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 155–168, 2017.

[8] Tobias Bleifuß, Leon Bornemann, Theodore Johnson, Dmitri V Kalashnikov, Felix Naumann, and Divesh Srivastava. Exploring change: a new dimension of data analytics. *Proceedings of the VLDB Endowment*, 12(2):85–98, 2018.

[9] Kevin Borders, Jonathan Springer, and Matthew Burnside. Chimera: A declarative language for streaming network traffic analysis. In *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*, pages 365–379, 2012.

[10] Jeffrey D Case, Mark Fedor, Martin L Schoffstall, and James Davin. Simple network management protocol (snmp). Technical report, 1990.

[11] Ang Chen, Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. The good, the bad, and the differences: Better network diagnostics with differential provenance. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 115–128. ACM, 2016.

[12] James Cheney, Laura Chiticariu, Wang-Chiew Tan, et al. Provenance in databases: Why, how, and where. *Foundations and Trends® in Databases*, 1(4):379–474, 2009.

[13] Sean Choi, Boris Burkov, Alex Eckert, Tian Fang, Saman Kazemkhani, Rob Sherwood, Ying Zhang, and Hongyi Zeng. Fboss: building switch software at scale. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 342–356. ACM, 2018.

[14] Cisco. Ip slas configuration guide, cisco ios release 12.4t. http://www.cisco.com/c/en/us/td/docs/ios-xml/ios/ipsla/configuration/12-4t/sla-12-4t-book.pdf.

[15] Mahmoud Elkhodr, Belal Alsinglawi, and Mohammad Alshehri. Data provenance in the internet of things. In *2018 32nd International Conference on Advanced Information Networking and Applications Workshops (WAINA)*, pages 727–731. IEEE, 2018.

[16] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Evolve or die: High-availability design principles drawn from googles network infrastructure. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 58–72. ACM, 2016.

[17] Nicolas Guilbaud and Ross Cartlidge. Google localizing packet loss in a large complex network. Nanog57, Feb 2013.

[18] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 139–152, 2015.

[19] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 357–371, 2018.

[20] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *NSDI*, volume 14, pages 71–85, 2014.

[21] Ahmed E Hassan and Richard C Holt. Predicting change propagation in software systems. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pages 284–293. IEEE, 2004.

[22] Qun Huang, Xin Jin, Patrick PC Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. Sketchvisor: Robust network measurement for software packet processing. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 113–126, 2017.

[23] Qun Huang, Patrick PC Lee, and Yungang Bao. Sketchlearn: Relieving user burdens in approximate measurement with automated statistical inference. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 576–590, 2018.

[24] Qun Huang, Haifeng Sun, Patrick PC Lee, Wei Bai, Feng Zhu, and Yungang Bao. Omnimon: Re-architecting network telemetry with resource efficiency and full accuracy. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 404–421, 2020.

[25] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Flowradar: A better netflow for data centers. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, pages 311–324, 2016.

[26] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. Nitrosketch: Robust and general sketch-based monitoring in software switches. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 334–350. 2019.

[27] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 101–114, 2016.

[28] Chris Lonvick. The bsd syslog protocol. Technical report, 2001.

[29] Jeffrey C Mogul, Drago Goricanec, Martin Pool, Anees Shaikh, Douglas Turk, Bikash Koley, and Xiaoxue Zhao. Experiences with modeling network topologies at multiple levels of abstraction. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 403–418, 2020.

[30] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Scream: Sketch resource allocation for software-defined measurement. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, pages 1–13, 2015.

[31] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Trumpet: Timely and precise triggers

[32] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 85–98, 2017.

[33] Srinivas Narayana, Mina Tahmasbi, Jennifer Rexford, and David Walker. Compiling path queries. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, pages 207–222, 2016.

[34] Yanghua Peng, Ji Yang, Chuan Wu, Chuanxiong Guo, Chengchen Hu, and Zongpeng Li. detector: a topology-aware monitoring system for data center networks. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 55–68. USENIX Association, 2017.

[35] Santhosh Prabhu, Kuan Yen Chou, Ali Kheradmand, Brighten Godfrey, and Matthew Caesar. Plankton: Scalable network configuration verification through model checking. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pages 953–967, 2020.

[36] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. Inside the social network's (datacenter) network. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 123–137. ACM, 2015.

[37] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, and Alex C Snoeren. Passive realtime datacenter fault detection and localization. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 595–612, 2017.

[38] Brandon Schlinker, Hyojeong Kim, Timothy Cui, Ethan Katz-Bassett, Harsha V Madhyastha, Italo Cunha, James Quinn, Saif Hasan, Petr Lapukhov, and Hongyi Zeng. Engineering egress with edge fabric: Steering oceans of content to the world. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 418–431. ACM, 2017.

[39] Yogeshwer Sharma, Philippe Ajoux, Petchean Ang, David Callies, Abhishek Choudhary, Laurent Demailly, Thomas Fersch, Liat Atsmon Guz, Andrzej Kotulski, Sachin Kulkarni, Sanjeev Kumar, Harry Li, Jun Li, Evgeniy Makeev, Kowshik Prakasam, Robbert Van Renesse, Sabyasachi Roy, Pratyush Seth, Yee Jiun Song, Benjamin Wester, Kaushik Veeraraghavan, and Peter

in data centers. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 129–143, 2016.

Xie. Wormhole: Reliable pub-sub to support geo-replicated internet services. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, Oakland, CA, May 2015. USENIX Association.

[40] Peng Sun, Ratul Mahajan, Jennifer Rexford, Lihua Yuan, Ming Zhang, and Ahsan Arefin. A network-state management service. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 563–574, 2014.

[41] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky HY Wong, and Hongyi Zeng. Robotron: Top-down network management at facebook scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 426–439. ACM, 2016.

[42] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Simplifying datacenter network debugging with path-dump. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 233–248, 2016.

[43] Cheng Tan, Ze Jin, Chuanxiong Guo, Tianrong Zhang, Haitao Wu, Karl Deng, Dongming Bi, and Dong Xiang. Netbouncer: active device and link failure localization in data center networks. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 599–614, 2019.

[44] Mea Wang, Baochun Li, and Zongpeng Li. *sFlow: Towards resource-efficient and agile service federation in service overlay networks*. IEEE, 2004.

[45] Yang Wu, Ang Chen, and Linh Thi Xuan Phan. Zeno: Diagnosing performance problems with temporal provenance. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 395–420, 2019.

[46] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve

Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 561–575. ACM, 2018.

[47] Da Yu, Yibo Zhu, Behnaz Arzani, Rodrigo Fonseca, Tianrong Zhang, Karl Deng, and Lihua Yuan. dshark: a general, easy to program and scalable framework for analyzing in-network packet traces. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 207–220, 2019.

[48] Minlan Yu, Lavanya Jose, and Rui Miao. Software defined traffic measurement with opensketch. In *NSDI*, volume 13, pages 29–42, 2013.

[49] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. Automatic test packet generation. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 241–252. ACM, 2012.

[50] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y Zhao, et al. Packet-level telemetry in large datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 479–491, 2015.

[51] Thomas Zimmermann, Andreas Zeller, Peter Weissgerber, and Stephan Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.

APPENDIX

The first step of PCAT is to collect data from devices, which we call discovered data. There are three types of data including numeric counters, non-numeric states, and configurations. Table 4 shows the examples for each category.

Types	Categories & examples	Impact of software upgrades
Counters	*Device utilization:* CPU&memory utilization, routing table size, etc	Ambiguity between percentage and absolute values.
	Device internal status: Interface error counter, power supply temperature, fan speeds, linecard version, optical CRC error counter, etc	XML format gets changed; linecard version format changes from integer to string.
	Packet processing counters: Packet drops, errors, queue length, etc	Ambiguity of interface stats meaning.
	Protocol counters: BGP neighbor received routes, etc	General empty data error.
States	*Interface state:* Interface up, down, drained, configured IP address, MAC address, etc	Hex-decimal change causes MAC address retrieving error.
	Protocol state: BGP neighbor state, etc	State meaning ambiguity.
Configs	BGP policy, queuing algorithm, etc	Raw config format changed.

Table 4: Different discovered data in PCAT.

SwarmMap: Scaling Up Real-time Collaborative Visual SLAM at the Edge

Jingao Xu[1*], Hao Cao[1*], Zheng Yang[1✉], Longfei Shangguan[2]
Jialin Zhang[1], Xiaowu He[1], Yunhao Liu[1]
[1]*School of Software, Tsinghua University*   [2]*University of Pittsburgh & Microsoft*

Abstract

The Edge-based Multi-agent visual SLAM plays a key role in emerging mobile applications such as search-and-rescue, inventory automation, and industrial inspection. This algorithm relies on a central node to maintain the global map and schedule agents to execute their individual tasks. However, as the number of agents continues growing, the operational overhead of the visual SLAM system such as data redundancy, bandwidth consumption, and localization errors also scale, which challenges the system scalability.

In this paper, we present the design and implementation of SwarmMap, a framework design that scales up collaborative visual SLAM service in edge offloading settings. At the core of SwarmMap are three simple yet effective system modules — a change log-based server-client synchronization mechanism, a priority-aware task scheduler, and a lean representation of the global map that work hand-in-hand to address the data explosion caused by the growing number of agents. We make SwarmMap compatible with the robotic operating system (ROS) and open-source it[1]. Existing visual SLAM applications could incorporate SwarmMap to enhance their performance and capacity in multi-agent scenarios. Comprehensive evaluations and a three-month case study at one of the world's largest oil fields demonstrate that SwarmMap can serve $2\times$ more agents (>20 agents) than the state of the arts with the same resource overhead, meanwhile maintaining an average trajectory error of 38*cm*, outperforming existing works by >55%.

1 Introduction

Visual simultaneous localization and mapping (SLAM) systems take video streams from one or multiple cameras as input, reconstructing the 3D map of environment while simultaneously determining the position and orientation of cameras with respect to their surroundings [29, 34, 36]. With the size of the mapping area expanding rapidly, collaborative visual

SLAM that involves multiple agents has been attracting growing interest from both academia and industry [25, 39, 40, 49]. For instance, Amazon, JD, and Alibaba have deployed dozens of picking and sorting robots in their logistics warehouses to save labor cost [45]; DJI and Amazon have also been developing drone grouping and swarming technology for urban modeling, express delivery, and industrial inspection [12]. In these scenarios, each agent has to conduct not only the localization but mapping tasks in real-time due to (*i*) upper layer applications require the latest updated environment map to perform the subsequent maintenance and scheduling tasks, especially in those dynamic environments; and (*ii*) since the two modules are tightly coupled, an agent also relies on a high-quality on-board map for a better localization performance and vice-versa [3, 47].

The SLAM agents profile the environment with their cameras, exchange data with each other, and execute vision tasks in real-time, with a significant computation overhead. The limited computation resource on the agent soon becomes the bottleneck, impairing system accuracy [3, 40, 47]. *Edge-offload* has emerged as a promising alternative due to the following two reasons. First, by offloading bulky tasks to edge devices, the agents only need to run light-weight and time-sensitive jobs locally, which effectively mitigates on-board resource shortage [3, 47]. Second, by fusing and further optimizing the visual map globally at a centralized edge device, map information that is originally unavailable to each other can be easily shared among agents [39, 40]. This will benefit collaborative missions such as collision avoidance and path planning.

Albeit inspiring, the growing number of agents brings new issues that challenge the scalability of edge-based real-time collaborative visual SLAM systems (§2.2):

• **Map synchronization stresses the network bandwidth**. Mobile agents like drones and robots heavily rely on wireless links to communicate with an edge device. However, wireless spectrum is a limited and overcrowded resource. Streaming large volumes of map data over wireless links will soon saturate the medium and cause significant delays.

• **FCFS-based job scheduling impairs the localization ac-**

---

[1]Code and data at https://github.com/MobiSense/SwarmMap.

Figure 1: Industrial inspection is carried out by 10 drones and 2 autonomous vehicles in one of the world's largest oil-field ($>170km^2$) in the Middle East. These agents are coordinated by SwarmMap that runs on an Nvidia AGX Xavier edge server.

curacy. An edge device has to processes large volumes of requests from agents, which may cause significant delays to latecomers (i.e., those requests positioned in the tail of the queue). However, agents in different states are not equally sensitive to the queuing delay. The conventional first-come, first-served (FCFS) pipeline will exacerbate the localization error on those time-sensitive agents.

• **Map expansion exacerbates the memory footprint**. The size of the global visual map increases sharply with a growing number of agents, which is likely to exceed the limited memory capacity allocated to SLAM tasks by an edge node, causing memory overflow.

However, the current practice of edge-offload focuses primarily on computation-oriented task partitioning [3, 8, 23, 40, 47]. They fail to address the data explosion and its impact on transmission, scheduling, and storage. Hence these pioneer designs cannot scale with the sheer size of the real-time collaborative visual SLAM systems.

In this work, we present SwarmMap, a framework to scale up the real-time collaborative visual SLAM services at resource-constrained edge devices. SwarmMap does not innovate visual SLAM algorithms. Instead, it proposes functionality and resource abstractions of existing SLAM algorithms and provides additional system services to enhance system scalability. Hence, most variations of collaborative visual SLAM systems can take advantage of our design. With SwarmMap, the upper-layer user can outsource agent task scheduling and processing instead of understanding every detail of SLAM algorithms to manually adapt. SwarmMap contains three key plug-in modules, as described below.

First, we design a Map Information Tracker (*Mapit*) to maintain map data consistency between the agents and the edge while remarkably saving network bandwidth. Unlike existing methods that transfer bulky map data with each other [39, 40], *Mapit* records the operations associated with the map modification on the agent and transmits these operations to the edge. The edge node then follows these operations to update its local map. This allows the map synchronization

between them at the minimum bandwidth consumption even compared with state-of-the-arts (e.g., CarMap [2]).

Second, we introduce a SLAM-specific task-aware scheduler (*STS*) that prioritizes requests based on the status of their producer (i.e., agent). The *STS* scheduler runs on both the agent and the edge. The agent *STS* evaluates agent status around the clock and updates this information with the edge through heartbeat packets. The edge *STS* designs a multi-level queue to ensure those urgent tasks will be processed timely.

Third, we propose a Map Backbone Profiling (*MBP*) technique to alleviate the storage overhead while retaining the mapping accuracy. This technique is based on an observation that the data quality among different agents' maps can be balanced by elements in co-visible areas. We propose a set of metrics to detect high-quality map elements and use them to offset those low-quality counterparts, thereby elevating the overall map quality. Applying model compression to this high-quality map allows us to remove large portions of redundant map data without sacrificing the map accuracy.

We evaluate SwarmMap on a testbed consisting of 4 Nvidia Jetson boards, 4 smartphones, 4 DJI RoboMasters, and 4 drones. Following the standard SLAM evaluation pipeline [2, 6, 28, 47], we further compare SwarmMap with two state-of-the-art (SOTA) edge-assisted multi-agent SLAM systems (CCM-SLAM [40] and Multi-UAV [39]) on three gold-standard SLAM datasets (TUM [11], KITTI [10], and EuRoC [9]) as well as a self-labeled dataset collected at a 22,927 sqft shopping mall. We also compare SwarmMap with CarMap [2] and Sum-Map [27] to evaluate each functional module in SwarmMap. Our head-to-head comparison shows that SwarmMap can serve $2\times$ more agents than these SOTA systems with the same resource overhead, meanwhile maintaining an absolute trajectory error within $38cm$ when serving 20 agents, outperforming these SOTA systems by $>55\%$.

Real-world deployment. We have developed a real-time collaborative visual SLAM system based on SwarmMap and deployed it in one of the world's largest oil-field ($>170km^2$) for industrial inspection (shown in Fig. 1). Our system con-
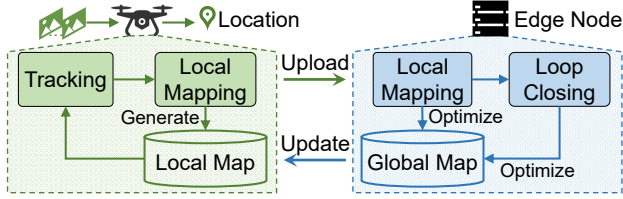
Figure 2: Workflow of existing edge-assisted SLAM [3, 47]

sists of twelve agents that communicate with an Nvidia Jetson AGX Xavier [46] edge node through Wi-Fi mesh networks. A three-month pilot study shows that SwarmMap achieves an average localization accuracy of 0.36*m*. The link throughput and RAM consumption are below 17MB/s and 26GB respectively, meeting inspection demands within the constraints of available resources.

In summary, this paper makes three contributions. First, we quantify the scalability challenges of deploying real-time collaborative visual SLAM at the edge to motivate framework support. Second, we design and implement SwarmMap as a framework to address the scalability issues spanning from communication, computation, to storage. As far as we are aware of, SwarmMap is the first system solution to scale up the collaborative visual SLAM in edge settings. Third, we deploy SwarmMap in one of the world's largest oil fields for industrial inspections in the Middle East. Our three-month pilot study demonstrates that SwarmMap makes a great process towards fortifying multi-agent collaborative visual SLAM to a fully practical system for wide deployment.

Contribution to the community. We implement SwarmMap as a software package of the robot operating system (ROS [26]), the dominating OS in the robotics field. We believe SwarmMap can provide a collection of tools for both academia and industry, and further enable fast prototyping of visual SLAM-based applications in multi-agent scenarios.

2   Background and Motivation

The data volume scales with the number of agents, and the need for framework support arises from the excessive bandwidth consumption and memory footprint caused by the data explosion. We discuss these in detail in this section.

2.1   Edge-assisted visual SLAM systems

The visual SLAM consists of multiple sub-tasks with diverse workloads. Edge-offload places those bulky tasks to an edge server, leaving an agent light-weight and time-sensitive jobs. The agent can thus run visual SLAM in real-time. We use ORB-SLAM2 [29], a top-ranked open-source visual SLAM system, to illustrate the SLAM operations under edge settings (refer to Fig. 2).

Front-end. Mobile agents run *Tracking* and part of the *Local Mapping* module locally. The *Tracking* module extracts 2D ORB feature points from each video frame and instantly estimates the pose of onboard camera(s) based on the geometry relationship between these feature points and the

pre-constructed local map (i.e., a set of 3D map-points and keyframes[2] in which they appear). As the mobile agent moves, the *Local Mapping* module updates the local map timely.

Back-end. Due to high computation costs, the optimization part of the *Local Mapping* module is offloaded to the edge device, where the bundle adjustment (BA) algorithms [42] kick in to improve the pose and 3D location accuracy of those newly generated keyframes and map-points. The edge server also runs a *Loop Closing* module to detect repeated paths and leverage them to re-calibrate the global map.

Data transfer in-between. To improve the map accuracy, each agent periodically sends keyframes and map-points to the edge server for fine-grained optimization. The optimized visual map is then streamed to the clients.

2.2   The scalability issues

As more agents get involved, running real-time collaborative visual SLAM on edge environment becomes increasingly complex, facing several challenges: (*i*) the frequent data transfer between agents and edge is likely to saturate wireless links, causing significant delays; (*ii*) the queueing delay on edge node exacerbates localization errors; (*iii*) the data volume grows sharply, threatening the data storage at the edge node. We discuss these issues below.

C1: Excessive bandwidth consumption. The life-cycle of a collaborative visual SLAM system consists of cold-start and maintenance two sessions. In the cold-start session, the agents transfer all observed keyframes and map-points data to the edge server. The edge server then generates a global map of the entire space and optimizes the local map for each agent. Once the global map generation has been completed, the SLAM system enters the long-term maintenance session during which each agent regularly revisits each site and calibrates the mapping offset. However, since map elements are tightly coupled, a minor modification on a single map element will spread to many other elements. This will cause a significant amount of data transfer in the maintenance.

To reduce bandwidth consumption, recent works [2, 40] design compact map representations and transfer the difference before and after map element calibration (as opposed to transferring the entire calibrated map element [39, 47]). Although these systems can effectively reduce bandwidth consumption in the cold-start session, they encounter two issues in the maintenance session due to the frequent map updates: (*i*) *extra computation overhead*. The acquisition of element-level differences requires pair-wise map feature comparison across the entire map. This will lead to extra computation workload pressure on resource-limited mobile agents; and (*ii*) *limited data volume reduction*. Since a minor change on an element will spread to a batch of coupled elements. the volume of data to be transferred is still bulky.

---

[2]Keyframes are a subset of selected frames. Each keyframe stores the camera pose, the map-points it observed, and the co-visibility relationships with other keyframes.

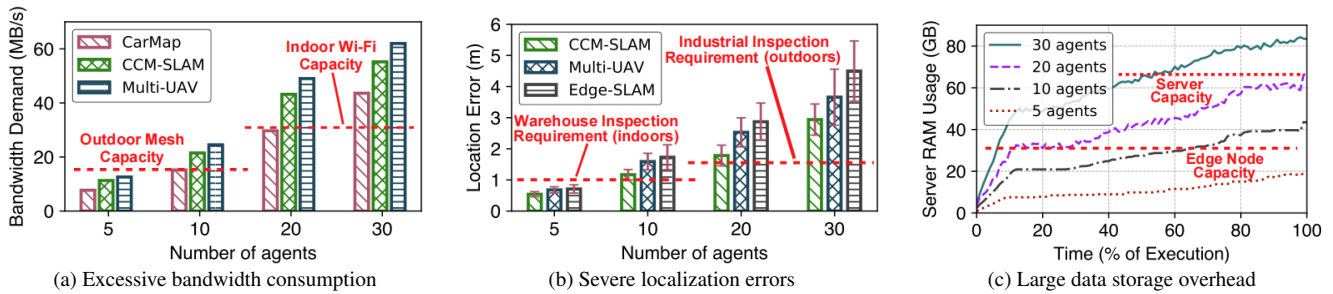(a) Excessive bandwidth consumption   (b) Severe localization errors   (c) Large data storage overhead

Figure 3: The scalability issues in multi-agent scenarios.

To validate our analysis, we measure the bandwidth requirement of three state-of-the-art (SOTA) systems in different number of agents settings. The results are shown in Fig. 3a. Compared with the vanilla Multi-UAV [39], we observe that CarMap [2] and CCM-SLAM [40] can effectively reduce the transmission workload during map synchronization. However, when serving more than ten agents, both systems still produce excessive wireless traffic that can easily go beyond the link capacity[3]; thus, significant system delays are expected.

C2: Severe localization errors. Under the edge settings, the localization accuracy of an agent highly depends on the quality of the local map which is optimized at the edge side. Typically, an agent needs to periodically (within 5s) send optimization requests to the edge server for every 3-5 newly generated keyframes [3, 47]. As the number of agents scales, the concurrent requests from different agents block at the edge node's processing pipeline, resulting in excessive queuing delays. Consequently, some agents get their optimization tasks done untimely, causing severe localization errors. This situation is worsened by the fact that agents in different running states (e.g., flying speeds, self-tracking qualities) are not equally sensitive to the waiting delay. Recent multi-agent collaborative SLAM solutions focus on map fusion and optimization on edge or cloud servers, but ignore the task queuing issue for each agent. The conventional first-come, first-served (FCFS) scheduling will inevitably exacerbate the localization error on those task-sensitive agents (demonstrated in §5.3).

We measure the localization error (in *m*) of three related works in a different number of agents settings. The results are shown in Fig. 3b. Considering the accuracy requirement from a broad range of SLAM applications, we treat 1*m* and 1.5*m* as acceptable localization errors for indoor (warehouse inspection) and outdoor (anomaly detection) scenarios. Evidently, all these three systems fail to meet the localization requirement when serving more than 5 and 10 agents indoors and outdoors respectively, leaving room for improvements.

C3: Large data storage overhead. The global map maintained by the edge server contains large redundancy due to the following two reasons. First, to ensure the inspection efficacy, different agents will re-visit the same area at certain intervals, causing significant path duplication. Second, to

complete the 3D map reconstruction, different agents have to share a co-visible area, resulting in bulky data redundancy. As the number of agents grows, the data redundancy increases sharply, and the data volume is likely to exceed the limited memory capacity of the edge node.

We set up an edge-based collaborative visual SLAM testbed using a commercial edge device Nvidia Jetson AGX Xavier (with 32GB RAM and costs $599) and measure its RAM usage in different numbers of agent settings. We repeat the measurement on a powerful server with 4× higher storage capacity (i.e., Dell PowerEdge T630 with 128GB RAM and costs $6,899) for comparison. The results are shown in Fig. 3c. In accordance with our analysis, as the system proceeds, the RAM usage increases rapidly and soon saturates the memory capacity of both the edge node and the high-end server. This limitation is worsened by the mismatch between the limited storage capacity of the edge node and the growing fidelity of video streams (i.e., 4K or 8K videos). Such high memory demand limits the maximum number of agents to five, which sets a strong barrier for the practical deployment of the edge-based collaborative visual SLAM system.

Due to the device heterogeneity (e.g., cameras on drones and robots may differ drastically in video resolution and frame rate) and diverse running status, the quality of maps provided by different agents may vary largely. An ideal map compression should remove those low-quality redundancy while retaining the high-quality counterpart. However, existing works ignore such difference when compressing the map data [27, 32, 43], resulting in degraded SLAM performance (details in §5.3).

2.3 SwarmMap: System goals

SwarmMap takes a solid step forward in solving these scalability issues. We list the system goals below.

Goal 1: Functionality and resource abstraction. SwarmMap should provide functionality and resource abstractions of existing SLAM algorithms. This allows any variation of map-point- and keyframe-based collaborative SLAM algorithms to take advantage of SwarmMap.

Goal 2: Plug and play. SwarmMap should be implemented as a plug-in module, exposing well-defined APIs to end-users for adaption. This avoids the deeply embedded manual code changes that may again challenge the system's scalability.

---

[3]The measurement shows the maximum throughput in an outdoor mesh and an indoor 2.4 GHz Wi-Fi network is 15MB/s and 30MB/s, respectively.

Goal 3: Resource overhead reduction. SwarmMap should effectively reduce the resource overhead spanning data storage, client-edge communication, and task scheduling while ensuring the precision and real-time performance.

3 Design

In this section, we first describe the high-level system architecture and then present each module design in SwarmMap.

3.1 System overview

SwarmMap is a framework design to scale up collaborative visual SLAM service in edge offloading settings. To achieve this goal, we make the following layer-wise functionality and resource abstractions: (*i*) *agent layer*, where each agent localizes itself and builds surrounding local maps in real-time; (*ii*) *network layer*, which enables communications and data interactions between mobile and edge for map synchronization; and (*iii*) *edge layer*, which fuses, optimizes, and maintains the global map. This layer-wise abstraction provides a clear view of map data transfer, processing, and storage in SLAMs.

Key functional modules. SwarmMap designs three plug-in modules to address the resource overhead and scheduling issues across these three layers.

• The *Mapit* (Map Information Tracker) module tracks system operations associated with map data calibration. It then transfers these operations to the peer(s) for map synchronization (§3.2).

• The *STS* (SLAM-specific Task Scheduling) module optimizes the batch request execution and manages the resource allocations among multiple agents (§3.3).

• The *MBP* (Map Backbone Profiling) module compresses the map data uploaded by individual agents while ensuring the overall mapping accuracy (§3.4).

SwarmMap Architecture. Fig. 4 shows the system architecture. SwarmMap shares similar edge-based architecture with previous works and provides extra system support on both the mobile agent and edge server side, as discussed below.

• On the mobile agent side, SwarmMap tracks the run-time status of each agent through a light-weight evaluation-based mechanism *STS* (mobile part). It then follows a dedicated information exchanging protocol *Mapit* to communicate and update map elements with the edge server.

• On the edge side, the edge node prioritizes the agents' requests by *STS* (edge part) based on their run-time status. It then takes into account the data quality of maps reported by individual agents and extracts a lean presentation of the overall map through a map backbone profiling algorithm (*MBP*). Finally, the optimized and compressed map backbones will be sent to each mobile agent by *Mapit*.

3.2 Mapit: Map Information Tracker

The inevitable frequent map data synchronization between clients and edge consumes large bandwidth in both cold-start
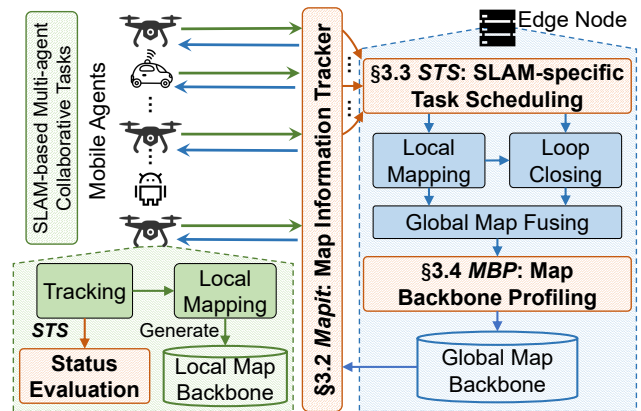

Figure 4: System architecture of SwarmMap. Compared with the conventional edge-based visual SLAM architecture, the added plug-in modules are highlighted in orange.

and maintenance sessions, circumscribing the system capacity (i.e., the number of supported agents). Recent works (e.g., CarMap [2] and CCM-SLAM [40]) propose a compact map representation that greatly reduces the data transfer in the cold-start session. However, their effectiveness fails to translate to a sufficient reduction in the maintenance session (§2.2-C1). Therefore, in SwarmMap we focus on the data transfer reduction in the maintenance session.

Our design is based on an observation that the map change on one side can be reproduced on the other side (e.g., agent vs. edge) by solely transferring the map change operations. This enables a light-weight map synchronization by avoiding transferring massive map-point data and the bulky geographical descriptors such as their spatial locations, features, observation relationships with keyframes [28]. Compared with the current practice, our design also achieves higher synchronization efficiency because it does not require a pair-wise map element comparison, which leads to extra computation workload pressure on resource-limited mobile agents.

To realize this basic idea, we design *Mapit*, a light-weight map information tracker to automate the operation tracking and reproducing on mobile and edge. *Mapit* runs as a daemon on both sides, monitoring the SLAM function calls and logging corresponding map operations (e.g., move a map-point by 2*cm*). It then transfers this log to the agent (or the server), based on which the agent reproduces these operations locally. The map data are synchronized at the end.

The *Mapit* package periodically[4] synchronizes the map operation logs, and consists of five atomic operations: *add*, *aggregate*, *push*, *merge*, and *pull* (shown in Fig. 5).

① *Mapit* **add.** The atomic operation *add* registers a hook for each SLAM function call (listed in Table 3) and maintains a recording queue. Whenever an important function is called, an operation record containing its name, parameters, and influence on map elements is *added* to the operation queue.

---

[4]Similar to current practice [2, 3, 39], we empirically set the period to 2s.
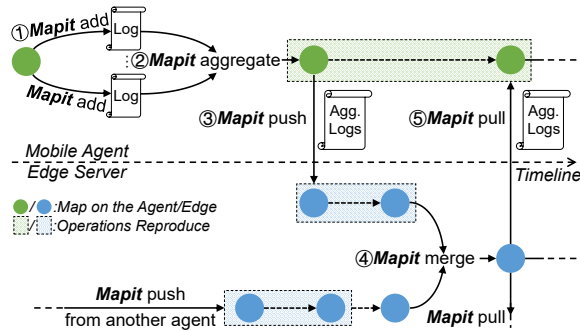
Figure 5: Workflow of the *Mapit*.

② ***Mapit* aggregate.** At the end of each period, *Mapit* aggregates the records in the operation queue to reduce their size. The intuition is that some removals or merges on certain types of functions will generate equivalent effects. For instance, if a function changes the location of a map-point and is marked as `overwritten`, we only need to focus on the latest record of it and ignore all the previous operations on the map-point. As for those marked as `stackable`, the implication is that records about modifying a same element can be merged by parameters. In this way, *Mapit* produces a minimal set containing necessary information.

③ ***Mapit* push.** After aggregating records, the atomic operation *push* on an agent sends the packed records to the edge server. By reproducing these operations, the map maintained on edge keeps synchronized with the ones on the client.

④ ***Mapit* merge.** On the edge server, the *merge* module periodically checks if there exists an overlap between the maps uploaded by individual agents and the global map. Once an overlap is detected, different maps will be coordinated and fused by the upper-layer SLAM algorithms (e.g., Sim3 optimization algorithm [28]). The map fusion process will operate and update some map elements, and hence the *merge* module also records these operations on the map elements in the same way as *add* and *aggregate*.

⑤ ***Mapit* pull.** The *pull* module can be treated as the reverse operation of *push*. It requests aggregated map modification logs generated by map optimization and *merge*, from the edge server to the agent. Additionally, if the global map has already been created (i.e., the whole system is in the maintenance session), *Mapit* will also transfer a set of closest map-points (e.g., associated with the next 5 keyframes) to the agent in the *pull* process. The benefit of this strategy is to enhance the agent's localization performance since these map-points with a high probability of appearing in the future would provide prior information for the *tracking* module on the agent side.

3.3   STS: SLAM-Specific Task Scheduling

As more agents get involved in SLAM systems, processing agents' requests (e.g., local map optimization) can cause excessive queuing delays. Since agents in different running states are not equally sensitive to the waiting delay, conventional FCFS scheduling may exacerbate localization errors

on time-sensitive agents and hurt SLAM performance (§2.2-C2). To our best knowledge, there is still a lack of scheduling strategy tailor to multi-agent SLAM tasks.

To address this issue, we introduce *STS* – the first SLAM-Specific Task Scheduler that guides the edge to strategically prioritize requests. Specifically, *STS* divides agents into emergency and non-emergency groups based on the agents' status. It timely reorders the requests based on the following principles:

(*i*) Prioritizing requests from agents in the emergency group.

(*ii*) Among those non-emergency agents, *STS* prioritizes requests from agents that can provide higher information gain for global map construction or optimization.

The first principle aims to prevent each agent from losing self-tracking, and the second is for achieving a better overall global mapping performance. We propose a set of metrics to characterize the agent status and design a multi-level queue to schedule the requests from agents.

3.3.1   Agent Status Evaluation and Updating

Agent side. Each agent regularly updates its status with the edge by sending heartbeat packets. Since both environment and device dynamics may fluctuate violently during an agent's movement, the heartbeat interval should be shorter than the agent's request interval (i.e., 2s). In SwarmMap we expose the heartbeat setting (100ms by default) to end-users so that they can easily adapt to different environment settings. We define three variables that can fairly reflect an agent's status:

• *Tracking state*: a 1-bit Boolean value shows whether an agent is traceable or not. An agent's tracking state is set to LOST if its latest ORB feature maps cannot well match the local feature map. This variable is provided by the *tracking* module in many visual SLAM systems [29].

• *Velocity burst*: a 1-bit Boolean value shows whether an agent's speed changes abruptly or not. An abrupt change of velocity may result in motion blur in videos and make it hard for clients to extract visual features. In SwarmMap, we set the variable *Velocity burst* to True if the current moving speed is 20% greater than the averaged speed over the latest N frames, where N is a variable exposed to end-users. $N = 10$ by default.

• *Tracked map-points number*: an 8-bit variable represents the number of map-points observed by an agent. A larger number indicates the *tracking* module is running more stable.

Server side. Due to the heterogeneous device capability (e.g., cameras on different agents may differ in resolutions) and diversified trajectory, each agent contributes unequally to global map construction and optimization. SwarmMap prioritizes requests from those agents that can provide higher information gain for global map construction and optimization. To this end, we design the following two metrics to measure the information gain of each agent:

• *Map-point score* (MS) is defined as the average score of all map-points observed by an agent (the way to calculate the map-point score will be introduced in §3.4). A higher average
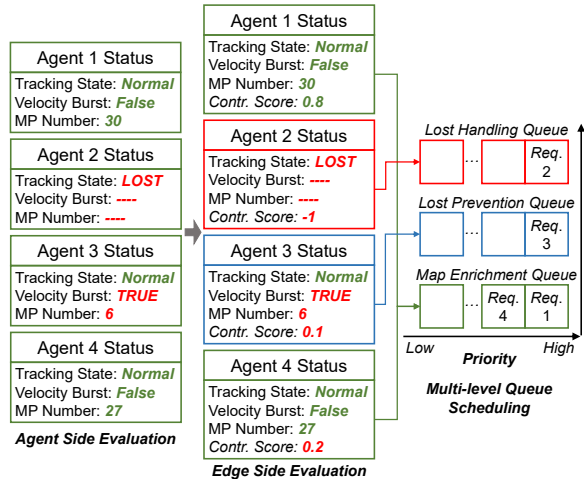
Figure 6: Workflow of the *STS* with an example.

score reflects that the current position is likely to have been visited before. On the contrary, a lower score indicates the agent is exploiting new or partially observed areas. Hence, *STS* prioritizes tasks with a lower map-point score.

• *Map elements generation speed* (MG) characterizes the number of unobserved map-points and keyframes uploaded by the latest *mapit push* operation. An agent with a higher map element generation speed contributes more to the edge's global map generation and optimization.

STS normalizes each metric and computes each agent's *contribution score* as normalized MG - normalized MS.

3.3.2 Multi-level Queue Scheduling

On the edge side, *STS* designs three queues with different priorities to facilitate agent request scheduling.

• **Lost Handling Queue**. If an agent's tracking state is marked as LOST, *STS* will push its request into this queue.

• **Lost Prevention Queue**. If an agent has a velocity burst and merely tracks few map-points, it may become prone to LOST, and *STS* will push its request into this queue.

• **Map Enrichment Queue**. For those agents with stable running status (i.e., without the risk of losing self-tracking), *STS* will push their requests into this queue and sort them by their mapping contribution scores.

The lost handling queue owns the highest priority, followed by lost prevention queue and map enrichment queue. Upon the reception of an agent's request, *STS* inserts this request into one of these three queues based on the agent's tracking status and mapping contribution. The back-end SLAM algorithm pops requests from queues based on their priority.

We take Fig. 6 as an example to explain the job scheduling in SwarmMap. Suppose there are four agents in the system, with agent 2 in lost tracking status and agent 3 facing the velocity burst issue. *STS* will push agent 2 and 3's requests into the lost handling and prevention queue, respectively. The request from agent 1 and 4, two agents not in emergency states, will be pushed into the map enrichment queue. Since

agent 1's mapping contribution score is higher than agent 4, the request from agent 1 will be put at the head of the queue. The edge processes these requests in the order of 2-3-1-4.

3.4 MBP: Map Backbone Profiling

The global map maintained by the edge node contains large redundancy (§2.2-C3). Due to the device heterogeneity (e.g., the onboard cameras may differ in resolution and frame rate) and diverse running status, the quality of maps contributed by different agents may vary largely. Existing map compression works [6, 13, 14] ignore such difference, resulting in information loss and hence degraded performance. The relevant works, CarMap and CCM-SLAM, design lean map representations to reduce the transmitted data volume for a faster map synchronization. However, they still need to reconstruct the huge global map through these compact representations on both mobile agent and edge node. Therefore, the memory footprint remains high when more agents are connected.

To address this issue, we introduce a map backbone profiling (*MBP*) algorithm. Unlike the current practice, we do not greedily remove redundant map elements in co-visible areas. Instead, we first leverage these redundant elements to generate a series of virtual keyframes and use them to improve those low-quality map segments. Once the overall quality of the global map got improved, we can thus compress the global map without compromising the mapping quality.

MBP first evaluates the quality and importance of each map element. It then (*i*): finds high-quality map-points that could be leveraged to generate virtual keyframes; (*ii*): searches for low-quality map segments that need to be improved; and (*iii*): improves the overall map quality by inserting virtual keyframes to those low-quality map segments. Finally, *MBP* operates map compression on the balanced global map.

3.4.1 Map Element Evaluation

Map-point evaluation has been extensively studied in related works [14]. The gold-standard metrics include the observing path length, maximum observing distance, maximum observing angle, and mean re-projection error. We borrow these metrics (detailed in §A.2) to evaluate a map-point and propose three new metrics to adapt to collaborative scenarios:

• **Observed number** represents the number of keyframes, in which the map-point is observed, across the entire global map. A higher score indicates multiple agents can observe a map point over a long period.

• **Update frequency** is defined as the total number of times the map-point was modified or updated by all agents in the last round of *Mapit push* operations. Map-points with high update frequency suggest a potential hot spot in a trajectory.

• **Moving velocity** records the speed of a mobile device when it generates the map element. A higher score indicates a potential blurriness that may influence the stability of the map-point. We take its negative value to evaluate the map-point score.
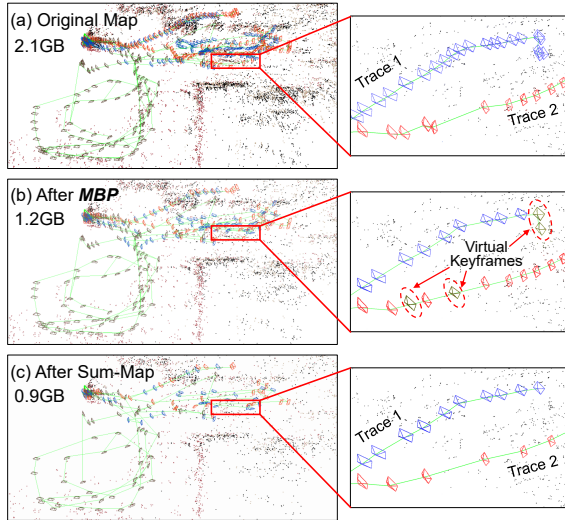
Figure 7: A running demo of *MBP*. The left column shows the map elements uploaded by different agents, while the right column presents partial zooming-in maps. The dotted keyframes in (b) are the synthetic virtual keyframes.

MBP normalizes each metric by its maximum value. We then define the score of a map-point as the sum of all normalized metrics values. The score of a keyframe is the sum of all observing map-point scores.

3.4.2 Map Backbone Generation

The map backbone generation consists of two steps: virtual keyframe generation and map compression.

Virtual keyframe generation. The trajectory of an individual agent is first segmented with the awareness of where overlaps occur. The quality of each map segment is defined as the sum of all map element scores (i.e., scores of all keyframes and map-points) within it. For each map segment with low quality (e.g., its score is in the bottom 20%), *MBP* search for high-quality map-points in its neighborhood (i.e., within 60° field-of-view of its keyframes) even though the original keyframes do not observe these map-points. Furthermore, *MBP* synthesizes virtual keyframes that could observe these high-quality map-points, and the pose (i.e., spatial location and orientation) of each keyframe can be calculated by the ICP algorithm [38] and optimized by BA [42]. Since the virtual keyframes only consider whether a map point is good enough regardless of which agent uploads it, they can supplement those low-quality segments.

Map compression. Once the quality of map segments is more balanced, *MBP* performs the similar map compression algorithm proposed by Sum-Map [27], eliminating redundancy by generating an enhanced minimum spanning tree across the global map. In addition, we introduce an extra optimization goal that guides the spanning tree to cover as many high-quality map elements as possible.

Fig. 7 compares the map compression performance of *MBP* and Sum-Map. Map elements from different agents are marked in red, blue, and brown in the figure. Although Sum-Map obviously reduces the map size, it neglects the map quality difference, making the compressed map of trace 2 too sparse and harming the SLAM performance. In contrast, with reducing the map size by nearly half, *MBP* inserts several virtual keyframes, balancing the map quality among different agents and ensuring mapping accuracy.

4 Implementation

We implement SwarmMap as an open-source package and make it compatible with ROS [26]. It contains 18,000 LOC (line of C++ code). SwarmMap is built upon ORB-SLAM2 [29], the top-ranked open-source SLAM algorithm that has been widely used by both research and industry communities. Our implementation avoids modifications on SLAM functions (e.g., tracking, local mapping, loop closing). This allows any variation of ORB-SLAM algorithms such as DynaSLAM [5], ORB-SLAM3 [7], as well as other map-point- and keyframe-based collaborative SLAM algorithms (e.g., Multi-UAV [39], C-ORB [22], CCM-SLAM [40]) to take advantage of SwarmMap (demonstrated in §A.5). Additionally, we also expose well-packaged APIs to facilitate users to modify some parameters (map synchronization period, status evaluation metrics, etc.) in SwarmMap according to specific upper-layer applications. A high-level abstraction of SwarmMap's implementation is detailed in §A.3.

5 Evaluation

In this section, we first present the experimental methodology (§5.1), followed by the overall performance of SwarmMap compared against SOTA systems (§5.2). We then conduct an ablation study to understand each functional module in SwarmMap (§5.3). Further, we demonstrate the portability of SwarmMap by plugging it into baseline SLAM systems (§A.5).

5.1 Experimental Methodology

Field studies. We deploy 12 agents including 4 smartphones, 4 drones, and 4 mobile robots on a 22,927 sqft shopping mall. These agents collaboratively localize themselves and mapping the environment in real-time. The ground truth is obtained through the Kinect 360 RGB-D and Opti-Track [33] cameras. We also build a dataset using these video streams for trace-driven evaluation.

Trace-driven evaluations. Following the conventional visual SLAM evaluation methodology [2, 22, 40, 47], we also conduct comprehensive trace-driven evaluations based on public SLAM datasets (KITTI [10], EuRoC [9], and TUM [11]) and the handcrafted dataset mentioned above. The characterization of three public datasets is summarized in Table 4. In our evaluations, the movement speed of mobile agents various significantly, ranging from 0.5m/s (indoor DJI RoboMasters) to 15m/s (ourdoor vehicles), representing the status of devices in

(a) ATE comparison on public datasets.   (b) Location error comparison on our dataset.   (c) Map updating latency comparison
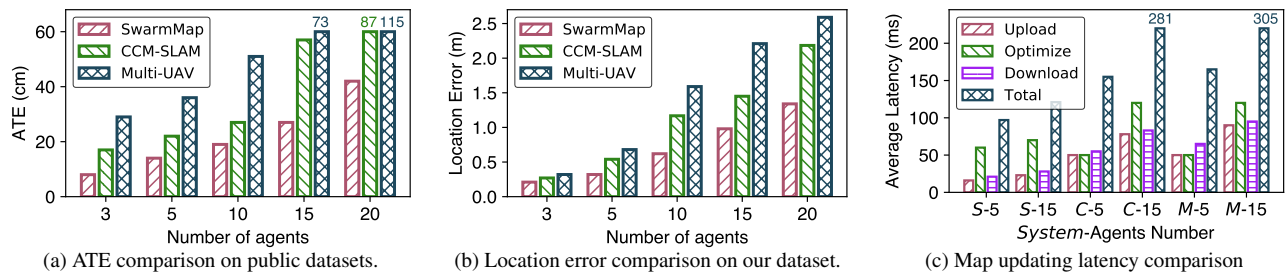
Figure 8: Overall performance comparison with a growing number of agents.

real world usage. Similar to the standard collaboration SLAM evaluation pipeline [19, 39, 40], we cut the video stream into overlapped segments and feed them to different agents to emulate the multi-agent scenario.

Edge Setup. Most of the previous works cannot be deployed on a resource-constrained edge node to support large numbers of agents because they consume a considerable amount of network bandwidth and edge computational resources (§2). We thus use a powerful server, which is equipped with an Intel(R) Xeon(R) CPU E5-2620v4 of 2.10GHz main frequency and 64GB RAM running Ubuntu 18.04, to explore the capacity of these systems and compare them with SwarmMap. The agents communicate with the server through 2.4 GHz and 5 GHz Wi-Fi links in the shopping mall and our laboratory. The maximally achievable link throughput measured with `iperf3` is 27.4MB/s and 46.1MB/s, respectively.

Metrics. We use *absolute trajectory error* (ATE, in *cm*) to evaluate SLAM accuracy on the three public datasets while adopting *location error* (in *m*) to evaluate the positioning accuracy in field studies and our handcrafted shopping mall dataset. ATE is a golden metric for evaluating the tracking performance of SLAM algorithms [11]. Since ATE pre-calibrates the generated trace with the ground-truth trajectories before measuring the absolute errors, it achieves fewer errors than the actual location errors. To evaluate system overhead, we count the *bandwidth demand* (in MB/s) of all participants in the system (defined as the sum of the average volume of data transferred per second by all agents). Similar to previous works [40, 47], we store the global map in RAM rather than SSD during system operation for faster map recall and update. We hence record the *RAM usage* (in GB) on the edge server to measure the memory consumption.

Map updating latency. Similar to previous works such as Edge-SLAM [3] and CCM-SLAM [40], SwarmMap adopts the same edge-assisted architecture where the *tracking* task is running locally on the agents. This allows an agent to localize itself in real-time (i.e., >30 fps with camera rate). We thus take the *map updating latency* (in *ms*)–the delay until the agent gets the latest optimized map from the server–as the metric to evaluate the real-time performance of map updating in SwarmMap. Map updating latency takes into account both the map synchronization and optimization latency.

5.2 Overall Performance Comparison

We first compare SwarmMap with CCM-SLAM [40] and Multi-UAV [39], two most relevant SOTA edge-based multi-agent SLAM systems, to evaluate the overall performance.

5.2.1 Accuracy Comparison

We first evaluate the average ATE and location error in a different number of agent settings. The results are depicted in Fig. 8a and Fig. 8b. As seen, SwarmMap achieves the best tracking and localization performance in all scenarios. Compared with related works, SwarmMap reduces ATE by > 30%, 20%, 20%, 50%, 55% for scales with 3, 5, 10, 15, 20 agents, respectively. The location errors are also significantly degraded by >40% when serving more than 10 agents. On the other hand, the performance of CCM-SLAM and Multi-UAV degrades remarkably with the growing number of agents. (i.e., the ATE and location errors expand 3× and 7× respectively from 3 to 20 agents). When serving more than 10 agents in the shopping mall, they fail to guarantee that the average location error of each client is within 1.5*m*, which is typically the localization precision requirement for indoor drones [48]. In contrast, SwarmMap can still bound ATE and location error within 40*cm* and 1.4*m* even serving 20 agents. Generally speaking, above delightful results come from the fact that the localization performance of each agent highly depends on the quality of the on-board maintained local map [3, 47], and the three modules (*Mapit*, *STS*, and *MBP*) in SwarmMap exactly enable each agent to acquire an optimized local map in time.

5.2.2 Map Updating Latency Comparison

We further examine the end-to-end latency of each agent from uploading map segments to eventually obtaining the optimized map from the edge node. To save space in the figure, we denote SwarmMap, CCM-SLAM, and Multi-UAV as *S*, *C*, and *M*, respectively. Fig. 8c shows the averaged latency on map uploading, optimizing, and downloading of each system in different number of agent settings. As seen, the total latency of SwarmMap is around 95*ms* and 105*ms* for 5 and 15 agents respectively, outperforming baselines by > 40% and 65%. The majority part of the latency reduction comes from the data uploading and downloading because *Mapit* reduces the amount of data transfers to a large extent. On the other hand,
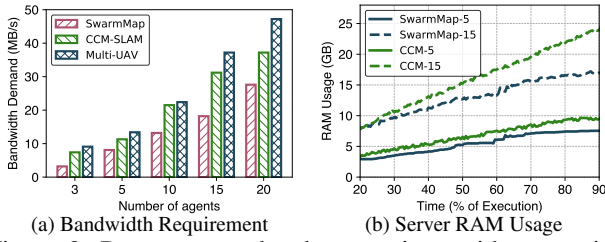
Figure 9: Resource overhead comparison with a growing number of agents.

Table 1: Transmitted data volume comparison.

Solution	Average Data Interaction Speed (MB/s)			
	TUM	KITTI	EuRoc	Shopping Mall
Mapit	**1.3**	1.1	1.3	**1.4**
CarMap	1.9	**0.9**	**1.2**	1.8
CCM-SLAM	3.2	1.9	2.2	2.9
Benchmark	5.2	4.3	4.7	4.9

Table 2: Map compression performance comparison.

Solution	KITTI 02		KITTI 05	
	Map Size (GB)	ATE (cm)	Map Size (GB)	ATE (cm)
MBP	3.1	**7.6**	1.9	**6.4**
Sum-Map	2.8	10.7	1.8	9.3
Benchmark	5.2	7.4	4.1	5.8

the processing latency also drops around 10% when serving 15 agents as *STS* module reduces the averaged queuing delay.

5.2.3 Resource Overhead Comparison

Bandwidth Demand. We then measure the bandwidth demand of these three systems. As depicted in Fig. 9a, on average, SwarmMap reduces > 35%, 20%, 30%, 25%, 20% of network bandwidth requirement when serving 3, 5, 10, 15, 20 mobile agents compared with existing works. Said differently, SwarmMap could serve more agents with the same wireless link throughput. For instance, under 27.4MB/s shopping mall bandwidth limitation, SwarmMap can support more than 20 agents while existing works merely around 10.

RAM Usage. We stitch the 00-05 trajectories on the KITTI dataset to generate a trajectory with 16.2*km* length and conduct a 30min experiment to measure the RAM usage. As shown in Fig. 9b, compared to CCM-SLAM, SwarmMap saves an average memory overhead of 2GB and 6GB when serving 5 and 15 agents, respectively, and the map size becomes stable under an upper bound (as seen, 15GB when serving 15 agents) once the whole scene is well mapped. Unlike CCM-SLAM which requires transmission of a large volume of map elements, SwarmMap leverages *Mapit* and significantly reduces the bandwidth demand for map synchronization. In addition, the *MBP* module prunes the size of the global map maintained and optimized on the server, thus reducing the system overhead on computational resources.

Generally speaking, SwarmMap aims to scale the collaborative SLAM service with the same resource overhead at the edge. SwarmMap will achieve a better performance with more computational resources are allocated and advanced resource management technologies (e.g., swap or virtual memory) are leveraged on edge, which are left as future works.

5.3 Ablation Study

We then conduct an ablation study to understand the effectiveness of each module in SwarmMap.

Performance of *Mapit*. We compare *Mapit* with CarMap [2], CCM-SLAM [40], and benchmark (e.g., edgeSLAM [47] and Edge-SLAM [3] that directly transmit the entire map without feature compression). Table 1 records the average data interaction speed (i.e., the average amount of map data uploaded and downloaded by each agent per second) of them

on different datasets. As seen, *Mapit* saves nearly two times the bandwidth compared to CCM-SLAM and benchmark on all datasets. *Mapit* performs slightly worse than CarMap on KITTI and EuRoc datasets, where the operating environments are relatively large (e.g., broad city roads). In these scenarios, the agents spend most of their time in the cold-start session during which they continuously transfer the newly generated map elements. In contrast, on TUM and our shopping mall datasets, the SLAM system completes the environment profiling quickly and soon enters the maintenance session during which *Mapit* eliminates map data transfer and saves the bandwidth by adopting the strategy of transmitting only records of map modifications rather than the modifications themselves.

Performance of *STS*. We evaluate *STS* by counting the average tracking lost percentage (i.e., proportion of video frames, with which agents fail to track themselves, in all video frames) of SwarmMap with (w/) and without (w/o) *STS*. As depicted in Fig. 10, despite the increasing service scale, SwarmMap (w/ *STS*) maintains a stable service quality, and the lost percentage is within 4% in all scenarios. In contrast, the lost percentage of CCM-SLAM as well as SwarmMap (w/o *STS*) increases rapidly, and the average lost percentage is at least 8% when serving more than 10 agents, which may lead to a terrible self-tracking and environmental mapping performance. Generally speaking, the *STS* strategy enables SwarmMap to prioritize tasks depending on the agent emergence states and prevent most agents from losing self-tracking.

Performance of *MBP*. We finally compare *MBP* with a map compression algorithm Sum-Map [27]. Specifically, we evaluate the map size after compression by their approaches and, equally important, the localization accuracy of each agent using the compressed map for self-tracking. The results are recorded in Table 2. We conduct experiments on the KITTI 02 and 05 trajectories because of the large map redundancy in them. The benchmark (only store the global map without compressing it) shows the size of the original map and the ATE by using it. As seen, *MBP* reduces the original map size by almost half. Although the map compression ratio of *MBP* is a little smaller than that of Sum-Map, *MBP* barely sacrifices the accuracy of the global map.
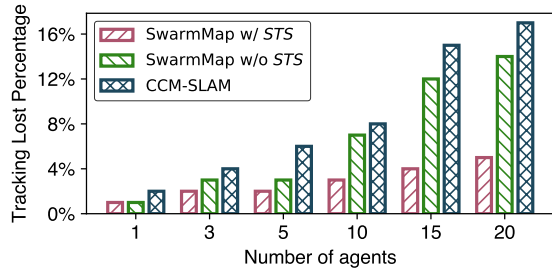
Figure 10: Tracking stability comparison.

6 Oil-field Case Study

Based on SwarmMap, we have developed a real-time collaborative visual SLAM system and deployed it in one of the world's largest oil-field ($> 170km^2$) in the Middle East for industrial inspection. The details about the deployment setups can be found in §A.6. We conduct a three-month pilot study (from June 2021 to August 2021) and summarize our main findings regarding the SLAM accuracy and system overhead.

SLAM Accuracy. We calculate the average location error of each agent during inspections and present these results in Fig. 11 and Fig. 12. Note that we cannot directly obtain ground-truth in the same way as in the experiment (e.g., deploy expensive Lidar or Opti-Track cameras), hence we collect the video frames captured by all agents and run the multi-agent ORB-SLAM3 offline afterward without considering the system latency. On this basis, we take the difference between the real-time localization performance of SwarmMap and offline processed results as the location error. As shown in Fig. 11, the average location error is 19.3cm and 29.1cm in indoor and outdoor scenarios, respectively, satisfying the task requirement (1m and 1.5m for indoor and outdoor inspections). Fig. 12 further illustrates the performance of each agent, and we find that two outdoor inspection drones (agents 9 and 10) suffer from a higher location error (up to 1m). The reason behind it is that these two drones are carrying out oil pipeline inspection at the border of the oil field; they fly faster (e.g., $> 5m/s$) and far away from the edge server (e.g., 15km). Therefore, they may experience certain delays due to the data forwarding through multi-hop mesh networks. Such a transmission delay may set a barrier for the drones to obtain the optimized map in time, causing localization errors. Nevertheless, the worst localization error of these two drones still satisfies the localization requirement in the outdoor scenario.

Latency. We measure each agent's onboard localization latency (the delay on estimating its own location from an input image) and map updating latency. The results are depicted in Fig. 13. We observe that each agent could localize itself in a real-time manner (i.e., the localization delay is within 35ms, typically the camera inter-frame interval). The average map updating delay is around 100ms. Although agent 9 suffers from a higher map updating delay (an average of 191ms) due to multi-hop data forwarding, it can still localize itself in real-time by leveraging its local map data.

Bandwidth demand. We record the total bandwidth demand

for indoor (4 agents) and outdoor (8 agents) inspection tasks. Fig. 14 shows a snapshot over a span of 175 minutes. We find there is a drop in bandwidth demand at 45min and 75min, respectively. This is because the SLAM system enters the maintenance session at these two time points. Thanks to *Mapit*, the transferred data volume in the maintenance session is significantly reduced, with 4MB/s for indoor and 11MB/s for outdoor inspections. Additionally, due to the relatively higher flight speed and map updating delay for outdoor drones, the edge server needs to frequently transmit updated maps to them in *Mapit pull* to prevent them from losing self-tracking, which results in the outdoor bandwidth demand fluctuates more dramatically than indoor ones.

RAM Usage. We further record the edge's RAM usage when executing the indoor and outdoor inspection tasks. As shown in Fig. 15, the maximum RAM usage in the indoor and outdoor scenarios is around 20GB and 12GB, both of which are well below the capability (32GB) of the edge node.

On-board CPU Usage. We also record the CPU occupancy rate of SwarmMap task (mobile part) on agent 1 (indoor drone) and 6 (outdoor drone) and plot these results in Fig. 16. The CPU usage of the outdoor drone is in the range of 20%-35%, while the indoor drone is 22%-43% during the 210 minutes of inspections. Due to the high dynamics of the indoor environment, the agent has to frequently update the local map although the whole area is well-mapped, which takes up more CPU resources than outdoor environments. Note that SLAM is an underlying algorithm that provides an agent with location and environmental information, and SwarmMap still leaves more than 50% CPU computational resources for each agent to perform upper-layer applications (e.g., context-aware interaction, object detection, or segmentation).

7 Related work

We review the most related works in this section.

Visual SLAM. One of the most fundamental algorithms in robotics has been a topic of research in robotics and mobile systems for several decades [6]. It consists of the concurrent construction of a surrounding environment and the state estimation of the robot moving within it. Typically, systems use monocular cameras [15, 20], stereo cameras [29], or RGB-D cameras [31]. Some of the more well-known visual SLAM examples include RGBD-SLAM [16], RTAB-Map [21], and ORB-SLAM [7,28,29]. Although SwarmMap is implemented on the top of ORB-SLAM2 [29], it can be easily ported to other map point-based visual SLAM like S-PTAM [34]. Other multi-map merging or optimization algorithms leveraged in recent work like ORB-SLAM3 [7], can also be integrated into SwarmMap. Our platform can also be applied to some feature/map point-based multi-sensor SLAM systems like VI-ORB [30], VINS [35], mmWave SLAM [24,44].

Edge-assisted Real-time SLAM. Recent studies [2, 3, 8, 23, 40, 47] speed up the computation-intensive tasks on agents by task partition and offloading workload to an edge server.
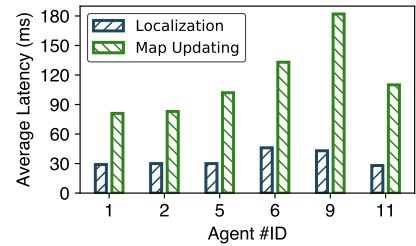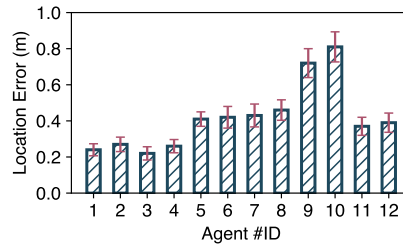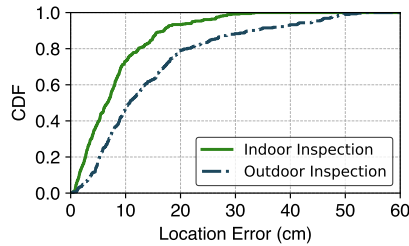
Figure 11: Different scenarios accuracy.

Figure 12: Accuracy of different agents.
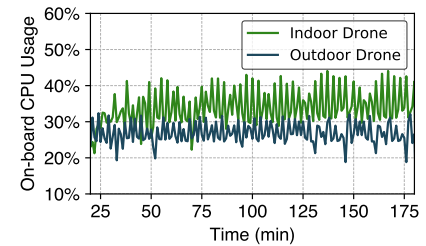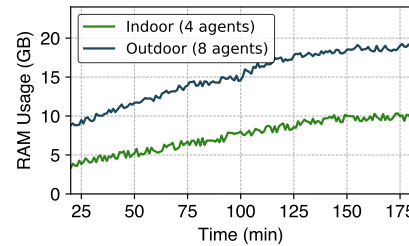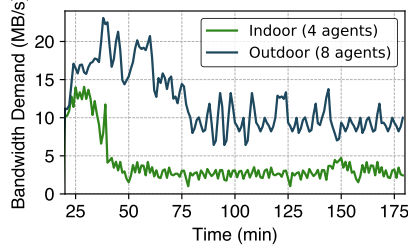

Figure 13: Latency measurement.

Figure 14: Bandwidth consumption.

Figure 15: Edge server RAM usage.

Figure 16: Mobile CPU occupation.

Therein, edgeSLAM [47] and Edge-SLAM [3] enable mobile agents to run visual SLAM in real-time. They split the original ORB-SLAM2 architecture and offload the *local mapping* and *loop closure* tasks to an edge server. CarMap [2] leverages the map constructed by crowdsourced agents and designs a near real-time map update framework between client and cloud. Muti-UAV [39] and CCM-SLAM [40] leverage a central server with potentially larger computational capacity to merge and optimize maps constructed by different agents, while each agent maintains partial local maps for tracking. However, as the number of serving agents scales, these works face severe scalability issues including excessive bandwidth consumption, severe localization errors, and large data storage. SwarmMap is the first work that solves these scalability issues based on the same edge settings.

Multi-agent Collaborative SLAM. Collaborative SLAM has been explored recently [6]. C2TAM [37], C-ORB [22], and CVI-SLAM [19] present collaborative SLAM frameworks based on PTAM [20], ORB-SLAM2 [29], and VI-ORB [7] respectively. CSfM [17] also proposes a framework to coordinate maps upload from different agents. In general, the system goals of these works and SwarmMap are orthogonal: above systems mainly focus on map fusion, optimization, and segmentation to generate a high-quality global map of the environment, ignoring the real-time performance of each agent and the entire system. In contrast, SwarmMap aims at solving the scalability issues and support each agent for real-time tracking, mapping, and map updating. Inspired by current efforts, we could integrate some map merging, optimizing, and even compressing algorithms proposed by recent works [6, 13, 14, 27, 49] into SwarmMap for a better SLAM performance, which are left as future works.

8   Discussion

We briefly discuss limitations and future work in this section. **The capacity of SwarmMap**. Although SwarmMap signif-

icantly reduces the bandwidth consumption and memory overhead for collaborative visual SLAM systems, such resource consumption still grows linearly with the number of the agents, which still fundamentally limits the system capacity. The way to make the resource consumption grow sub-linearly [18] with respect to the number of agents worth further research. On the other hand, the current *Mapit* design merely focuses on reducing bandwidth consumption in the maintenance session. Serving the system throughput the entire life-cycle with *Mapit* could potentially save more bandwidth. **Map optimization algorithms integration.** SwarmMap provides a basic map transmission and management platform for multi-agent SLAM. To date, SLAM map optimization is still a trending topic in the robotics field. Integrating existing advanced technologies (e.g., map compression, fusion, and semantic recognition) into SwarmMap for a better system performance is an ongoing work. Furthermore, efficient map data sharing not only between mobile and edge, but among different agents could also benefit upper layer applications.

9   Conclusions

We have presented the design and implementation SwarmMap, a framework to support real-time collaborative visual SLAM at edge devices. SwarmMap proposes functionality and resource abstractions of SLAM systems and provides three light-weight system services to address the communication, storage, and scheduling issues in edge-based scenarios. We implement SwarmMap as a software package on the ROS platform so that most variations of visual SLAM systems can directly benefit from it. Extensive evaluations and a three-month pilot study demonstrate its superior performance.

Acknowledgements

References

[1] Numba GPU Acceleration. https://numba.pydata.org/.

[2] Fawad Ahmad, Hang Qiu, Ray Eells, Fan Bai, and Ramesh Govindan. Carmap: Fast 3d feature map updates for automobiles. In *Proceedings of the USENIX NSDI*, 2020.

[3] Ali J Ben Ali, Zakieh Sadat Hashemifar, and Karthik Dantu. Edge-slam: edge-assisted visual simultaneous localization and mapping. In *Proceedings of the ACM Mobisys*, 2020.

[4] ArduPilot. https://ardupilot.org/ardupilot/.

[5] Berta Bescos, José M Fácil, Javier Civera, and José Neira. Dynaslam: Tracking, mapping, and inpainting in dynamic scenes. *IEEE Robotics and Automation Letters*, 3(4):4076–4083, 2018.

[6] Cesar Cadena, Luca Carlone, Henry Carrillo, Yasir Latif, Davide Scaramuzza, José Neira, Ian Reid, and John J Leonard. Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age. *IEEE Transactions on robotics*, 2016.

[7] Carlos Campos, Richard Elvira, Juan J Gómez Rodríguez, José MM Montiel, and Juan D Tardós. Orb-slam3: An accurate open-source library for visual, visual–inertial, and multimap slam. *IEEE Transactions on Robotics*, 2021.

[8] Tiffany Yu-Han Chen, Lenin Ravindranath, Shuo Deng, Paramvir Bahl, and Hari Balakrishnan. Glimpse: Continuous, real-time object recognition on mobile devices. In *Proceedings of the ACM Sensys*, 2015.

[9] EuRoC Dataset. https://projects.asl.ethz.ch/datasets/.

[10] KITTI Dataset. http://www.cvlibs.net/datasets/kitti/eval_odometry.php.

[11] TUM Dataset. https://vision.in.tum.de/data/datasets/rgbd-dataset/tools.

[12] DJI drones for industrial inspection. https://www.dji.com/products/industrial.

[13] M. Dymczyk, S. Lynen, M. Bosse, and R. Siegwart. Keep it brief: Scalable creation of compressed localization maps. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2536–2542, 2015.

[14] Marcin Dymczyk, Thomas Schneider, Igor Gilitschenski, Roland Siegwart, and Elena Stumm. Erasing bad memories: Agent-side summarization for long-term mapping. In *Proceedings of the IEEE IROS*, 2016.

[15] Jakob Engel, Thomas Schöps, and Daniel Cremers. Lsd-slam: Large-scale direct monocular slam. In *Proceedings of the Springer ECCV*, 2014.

[16] Nikolas Engelhard, Felix Endres, Jürgen Hess, Jürgen Sturm, and Wolfram Burgard. Real-time 3d visual slam with a hand-held rgb-d camera. In *Proceedings of the RGB-D Workshop on 3D Perception in Robotics at the European Robotics Forum, Vasteras, Sweden*, volume 180, pages 1–15, 2011.

[17] Christian Forster, Simon Lynen, Laurent Kneip, and Davide Scaramuzza. Collaborative monocular slam with multiple micro aerial vehicles. In *Proceedings of the IEEE IROS*, 2013.

[18] Samvit Jain, Xun Zhang, Yuhao Zhou, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, Paramvir Bahl, and Joseph Gonzalez. Spatula: Efficient cross-camera video analytics on large camera networks. In *Proceedings of the IEEE/ACM SEC*, 2020.

[19] Marco Karrer, Patrik Schmuck, and Margarita Chli. Cvi-slam—collaborative visual-inertial slam. *IEEE Robotics and Automation Letters*, 3(4):2762–2769, 2018.

[20] Georg Klein and David Murray. Parallel tracking and mapping for small ar workspaces. In *Proceedings of the IEEE ISMAR*, 2007.

[21] Mathieu Labbé and François Michaud. Rtab-map as an open-source lidar and visual simultaneous localization and mapping library for large-scale and long-term online operation. *Journal of Field Robotics*, 2019.

[22] Fu Li, Shaowu Yang, Xiaodong Yi, and Xuejun Yang. Corb-slam: a collaborative visual slam system for multiple robots. In *International Conference on Collaborative Computing: Networking, Applications and Worksharing*. Springer, 2017.

[23] Luyang Liu, Hongyu Li, and Marco Gruteser. Edge assisted real-time object detection for mobile augmented reality. In *Proceedings of the ACM Mobicom*, 2019.

[24] Chris Xiaoxuan Lu, Stefano Rosa, Peijun Zhao, Bing Wang, Changhao Chen, John A Stankovic, Niki Trigoni, and Andrew Markham. See through smoke: robust indoor mapping with low-cost mmwave radar. In *Proceedings of the ACM MobiSys*, 2020.

[25] Yunfei Ma, Nicholas Selby, and Fadel Adib. Drone relays for battery-free networks. In *Proceedings of the ACM Sigcomm*, 2017.

[26] ROS Melodic. https://wiki.ros.org/melodic.

[27] Peter Mühlfellner, Mathias Bürki, Michael Bosse, Wojciech Derendarz, Roland Philippsen, and Paul Furgale. Summary maps for lifelong visual localization. *Journal of Field Robotics*, 33(5):561–590, 2016.

[28] Raul Mur-Artal, Jose Maria Martinez Montiel, and Juan D Tardos. Orb-slam: a versatile and accurate monocular slam system. *IEEE Transactions on Robotics*, 31(5):1147–1163, 2015.

[29] Raul Mur-Artal and Juan D Tardós. Orb-slam2: An open-source slam system for monocular, stereo, and rgb-d cameras. *IEEE Transactions on Robotics*, 33(5):1255–1262, 2017.

[30] Raúl Mur-Artal and Juan D Tardós. Visual-inertial monocular slam with map reuse. *IEEE Robotics and Automation Letters*, 2(2):796–803, 2017.

[31] Richard A Newcombe, Steven J Lovegrove, and Andrew J Davison. Dtam: Dense tracking and mapping in real-time. In *Proceedings of the IEEE ICCV*, 2011.

[32] Van Opdenbosch et al. *Data Compression for Collaborative Visual SLAM*. PhD thesis, Technische Universität München, 2019.

[33] Opti-Track. https://optitrack.com/.

[34] Taihú Pire, Thomas Fischer, Gastón Castro, Pablo De Cristóforis, Javier Civera, and Julio Jacobo Berlles. S-ptam: Stereo parallel tracking and mapping. *Robotics and Autonomous Systems*, 93:27–42, 2017.

[35] Tong Qin, Peiliang Li, and Shaojie Shen. Vins-mono: A robust and versatile monocular visual-inertial state estimator. *Proceedings of the IEEE Transactions on Robotics*, 2018.

[36] Hang Qiu, Fawad Ahmad, Fan Bai, Marco Gruteser, and Ramesh Govindan. Avr: Augmented vehicular reality. In *Proceedings of the ACM MobiSys*, 2018.

[37] Luis Riazuelo, Javier Civera, and JM Martınez Montiel. C2tam: A cloud framework for cooperative tracking and mapping. *Robotics and Autonomous Systems*, 62(4):401–413, 2014.

[38] Szymon Rusinkiewicz and Marc Levoy. Efficient variants of the icp algorithm. In *Proceedings of the IEEE 3-D digital imaging and modeling*, 2001.

[39] Patrik Schmuck and Margarita Chli. Multi-uav collaborative monocular slam. In *Proceedings of the IEEE ICRA*, 2017.

[40] Patrik Schmuck and Margarita Chli. Ccm-slam: Robust and efficient centralized collaborative monocular simultaneous localization and mapping for robotic teams. *Journal of Field Robotics*, 36(4):763–781, 2019.

[41] CUDA Toolkit. https://developer.nvidia.com/cuda-toolkit.

[42] Bill Triggs, Philip F McLauchlan, Richard I Hartley, and Andrew W Fitzgibbon. Bundle adjustment—a modern synthesis. In *Proceedings of the Springer International workshop on vision algorithms*, 1999.

[43] Dominik Van Opdenbosch and Eckehard Steinbach. Collaborative visual slam using compressed feature exchange. *IEEE Robotics and Automation Letters*, 4(1):57–64, 2018.

[44] Teng Wei, Anfu Zhou, and Xinyu Zhang. Facilitating robust 60 ghz network deployment by sensing ambient reflectors. In *Proceedings of the USENIX NSDI*, 2017.

[45] Amazon Warehouse with Robots. https://www.wired.com/story/amazon-warehouse-robots/.

[46] Nvidia Jetson AGX Xavier. https://developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit.

[47] Jingao Xu, Hao Cao, Danyang Li, Kehong Huang, Chen Qian, Longfei Shangguan, and Zheng Yang. Edge assisted mobile semantic visual slam. In *Proceedings of the IEEE INFOCOM*, 2020.

[48] Shengkai Zhang, Wei Wang, and Tao Jiang. Wi-fi-inertial indoor pose estimation for microaerial vehicles. *Transactions on Industrial Electronics*, 68(5):4331–4340, 2020.

[49] Danping Zou, Ping Tan, and Wenxian Yu. Collaborative visual slam for multiple agents: A brief survey. *Virtual Reality & Intelligent Hardware*, 1(5):461–482, 2019.

A Appendix

A.1 Functions Registered in *Mapit*

In the *Mapit add* module, we dig the insights about how map elements get changed and find these changes mainly caused by certain important SLAM functions, a fraction of which is listed in Table 3. Thus, modifications that happened to the map can be recorded as calling history of these functions. For certain functions shown in the table, some removal and compression on the records will not harm data consistency. For instance, if a function is marked as overwritten, it indicates that its only effective change on a map element is the latest one i.e., changing the pose of a map point. As for those

Table 3: Functions that could change the map element (only some fundamental functions are listed)

Target	Function	Type	Description
KeyFrame	`SetPose`	overwritten	set the pose of the keyframe
KeyFrame	`AddMapPoint`	unique	add a map point to the keyframe
KeyFrame	`EraseMapPointMatch`	unique	remove a map point from the keyframe
KeyFrame	`SetBadFlag`	unique	mark the keyframe bad and delete it
MapPoint	`SetWorldPos`	overwritten	set map point position in the world coordinate
MapPoint	`AddObservation`	unique	add a keyframe that observes the map point
MapPoint	`EraseObservation`	unique	remove a keyframe from observations
MapPoint	`SetBadFlag`	unique	mark the map point bad and delete it
MapPoint	`IncreaseVisible`	stackable	increase the count that map point is observed
MapPoint	`IncreaseFound`	stackable	increase the count that map point is matched
MapPoint	`SetLastTrackedTime`	overwritten	set the last tracked time of the map point
MapPoint	`UpdateNormalAndDepth`	overwritten	update the normal vector and depth of the map point
Map	`Clear`	overwritten	clear the current map
Map	`AddLoopClosing`	unique	add a keyframe to loop closing queue

Table 4: Dataset Description

Dataset Label	Trajectory Sequence	Total Time (min)	Total Path (m)	Total Frames	Environment
T-M (TUM Medium & Easy)	fr2_desk	1.66	18.88	2965	office
	fr3_long_office_household	1.45	21.46	2585	
T-D (TUM Difficult)	fr2_large_with_loop	2.88	39.11	5182	industrial hall
	fr2_large_no_loop	1.87	10.93	3359	
K-M (KITTI Medium & Easy)	00 / 05	7.57 / 4.79	3724.18 / 2205.58	4541 / 2761	city road
K-D (KITTI Difficult)	02 / 04	7.77	5067.23 / 393.65	4661 / 271	city road
E-M (EuRoC Medium & Easy)	MH_01 / MH_02	2.47 / 2.50	68.52 / 73.50	3682 / 3040	machine hall
E-D (EuRoC Difficult)	MH_04 / MH_05	1.65 / 1.85	91.70 / 97.59	2033 / 2273	machine hall
Shopping Mall (Our Dataset)	N/A	15	314.2	24,365	shopping mall

marked `stackable`, the implication is that records about modifying the same element can be merged by parameters and still yield the same effect.

A.2 Map-point Evaluation Metrics

A typical SLAM map consists of two types of elements, map points and keyframes. Map points represent discrete 3D landmarks in the global coordinate, and keyframes are selected frames indicating poses and positions of the corresponding camera (as illustrated in Fig. 18 with corresponding notations in Table 5). EBM [14] introduces several features based on local geometry information; we list four important metrics to evaluate a map-point we used in *MBP*:

• **Observing Path Length**. The distance traveled while observing the map-point and is obtained as

$$\phi_d^i = \sum_{j \in S^i} \|\mathbf{t}_G^{j+1} - \mathbf{t}_G^j\|_2.$$

• **Maximum Observing Distance**. The distance traveled between two most distant keyframes on a track, and each of them observes the map-point. Its computation requires maximization over all keyframes observing the same map-point, i.e.,

$$\phi_\delta^i = \max_{j,k \in S^i} \|\mathbf{t}_G^j - \mathbf{t}_G^k\|_2.$$

• **Maximum Observing Angle**. The maximum angle between two keyframes that could observe the map-point and is obtained as

$$\phi_a^i = \max_{j,k \in S^i} \arccos(\mathbf{r}_G^{j,i} \cdot \mathbf{r}_G^{k,i}).$$

• **Mean Re-projection Error**. Apart from the map-point track geometry, it is also worth considering the consistency of the map in the map-point's locality. EBM calculate the average re-projection error of each map-point to represent the mapping stability, i.e.,

$$\phi_p^i = \frac{\sum_{j \in S^i} \|m_{i,j} - m'_{i,j}\|_2}{|S^i|}.$$

A.3 SwarmMap Abstraction

Fig. 17 shows the high-level abstraction of SwarmMap's implementation. The *MBP* module assists the map fusion and optimization unit to eliminate the data redundancy in the global map. The *STS* module replaces those handcrafted request handlers in conventional SLAM implementations [19, 39, 40] and thus alleviates the end users' development overhead. Finally, we replace the communication unit and map handlers with a unified *Mapit* module. Such a layered implementation decouples SwarmMap's functional modules, allowing the end-users to turn on/off each module as they need. It also avoids the deeply embedded manual code changes (e.g., defining handlers) that again challenge the system scalability.
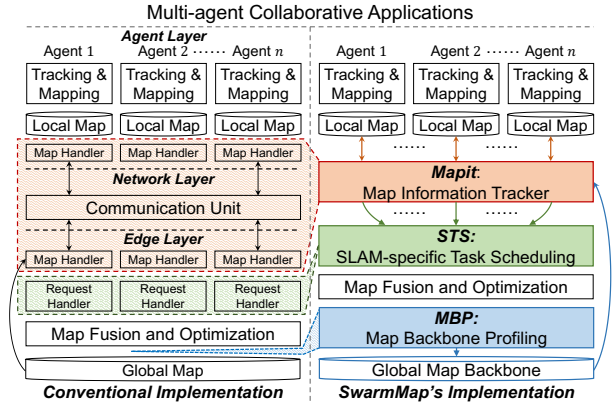
Figure 17: High-level abstraction of SwarmMap's implementation. The arrow shows the data flow.
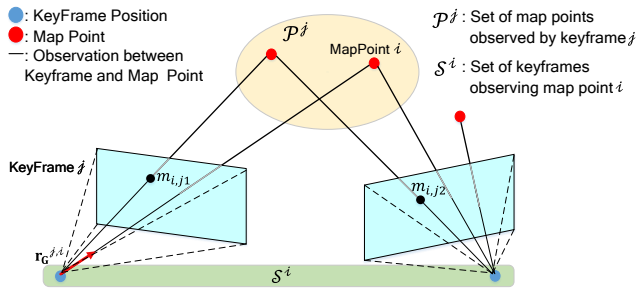

Figure 18: Observation connection between keyframes and map-points.

A.4 Experimental Dataset Description

We list the public datasets, the trajectories we used, and our handcrafted shopping mall dataset in Table 4. We select representative trajectories with different difficulty levels (in terms of environmental dynamics, path length, feature point sparsity, ambient light intensity, etc..) in TUM, KITTI, and EuRoc datasets, respectively.

A.5 Plug-and-play

We demonstrate the portability of SwarmMap by integrating each of its components into two different SLAM systems. We add *STS*, *Mapit*, and *MBP* to ORB-SLAM3 [7], the latest follow-up of the ORB-SLAM system, and measure the

Table 5: Notation Description

Notation	Description
$\mathbf{X}_{\mathsf{G}}^{i}$	position of map point i in global coordinate G
$\mathbf{t}_{\mathsf{G}}^{j}$	position of keyframe j in global coordinate G
$\mathcal{S}^{i}$	set of all keyframes observing map point i
$\mathbf{r}_{\mathsf{G}}^{j,i}$	unit-length observing vector starting from the observing keyframe j to map point i in global coordinate G
$\mathcal{P}^{j}$	set of all map points observed by keyframe j
$\mathcal{M}^{i}$	set of all agents observing map point i
t_c^i, t_l^i	creation and last tracked time for map point i
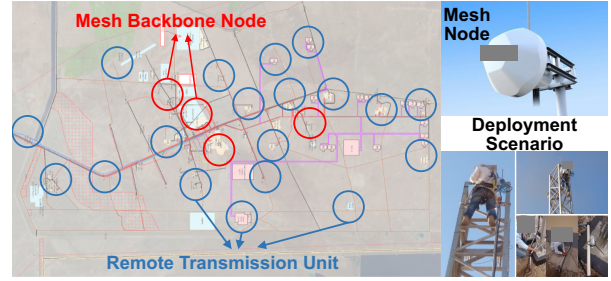
accuracy gain brought by each module. Fig. 20 shows the results. As seen, all these three modules contribute to localization accuracy. When serving 5 agents, *STS*, *Mapit*, and *MBP* decrease location errors by around 50% and 30%, and 10%, respectively. The contribution of each component also grows with an increasing number of agents.

We also integrate SwarmMap into our baselines CCM-SLAM, Multi-UAV, and ORB-SLAM3 (abbreviated as *C*, *M*, and *O*, respectively) to explore the location error reduction. As depicted in Fig. 21, the location error of CCM-SLAM, Multi-UAV, and ORB-SLAM3 decreases by 13.4%, 12.2%, and 16.7% respectively in 5 agents settings. When serving 15 agents, the error decreases further to 17.2%, 31.3%, and 29.6%.

Remarks. These results show that most existing works in multi-agent scenarios (especially scenarios with more agents) can directly benefit from SwarmMap. It is worth mentioning that we do not re-design or modify the code structure of these existing works for integration. We merely provide a wrapper to hook up these systems and SwarmMap (i.e., call the API defined in SwarmMap).

A.6 Case Study Setups

Our system consists of 12 mobile agents to perform daily inspection tasks both indoors and outdoors. These agents communicate with an Nvidia Jetson AGX Xavier edge node through Wi-Fi mesh networks, as shown in Fig. 19.

Inspection agents. We have deployed 12 mobile agents to perform daily inspection tasks, including 4 DJI Inspire drones (Agent #ID 1-4, equipped with 2K cameras) for indoor warehouse inspection as well as 6 DJI Inspire (#ID 5-10) and 2 inspection vehicles (#ID 11-12, equipped with 1080P cameras) for outdoor oil-field inspection. For drones, we integrate the mobile part of SwarmMap into ArduPilot [4], a widely-used open source drone development platform. The output localization and mapping results are streamed to the Ardupilot Mega controller through a Micro-USB port for supporting upper-layer applications (e.g., real-time drone flight control, abnormal events detection). The two inspection vehicles are equipped with Nvidia Jetson TX1 as their computing units.

Edge server. We implement the edge side of SwarmMap on an Nvidia Jetson AGX Xavier edge node with a 32GB 256-Bit LPDDR4x RAM, a 16-core ARM v8.2 64-bit CPU, and a

Figure 19: Mesh network deployment in the Oil-field.
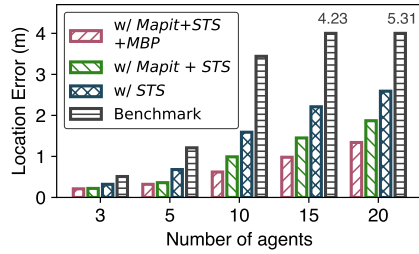
Figure 20: Performance of each module.

Figure 21: Performance gains.

512-core Volta GPU. We also turn on the GPU acceleration by Numba [1] and CUDA [41] to speed up the back-end global map optimization procedure. The power consumption of the edge node is below 30W, which is less than the available power supply in the industrial scenario.

Wireless Network. The 4 indoor inspection drones communicate with the edge node via 2.4 GHz WiFi, while the 8 outdoor inspection agents communicate through a mesh net-

work. In order to make the mesh network cover the whole $170km^2$ outdoor oil-field (the west-east distance is around $30km$), 24 communication nodes, including 4 mesh backbone nodes and associated 20 remote transmission units (RTU) are deployed (shown in Fig. 19). The maximum throughput measured by `iperf3` in the outdoor mesh and indoor WiFi network is 14.3MB/s and 26.8MB/s, respectively.

In-Network Velocity Control of Industrial Robot Arms

Sándor Laki[1], Csaba Györgyi[1], József Pető[2], Péter Vörös[1], and Géza Szabó[3]

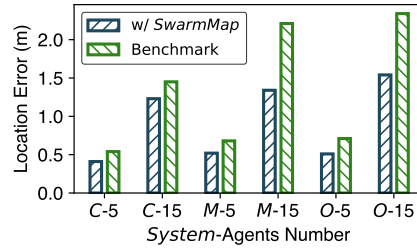[1]*ELTE Eötvös Loránd University, Budapest, Hungary*
[2]*Budapest University of Technology and Economics, Budapest, Hungary*
[3]*Ericsson Research, Budapest, Hungary*

Abstract

In-network computing has emerged as a new computational paradigm made possible with the advent of programmable data planes. The benefits of moving computations traditionally performed by servers to the network have recently been demonstrated through different applications. In this paper, we argue that programmable data planes could be a key technology enabler of cloud and edge-cloud robotics, and in general could revitalize industrial networking. We propose an in-network approach for real-time robot control that separates delay sensitive tasks from high-level control processes. The proposed system offloads real-time velocity control of robot arms to P4-enabled programmable data planes and only keeps the high-level control and planning at the industrial controller. This separation allows the deployment of industrial control in non-real-time environments like virtual machines and service containers running in a remote cloud or an edge-computing infrastructure. In addition, we also demonstrate that our method can smoothly control 100s of robot arms with a single P4-switch, enables fast reroute between trajectories, solves the precise synchronization of multiple robots by design and supports the plug-and-play deployment of new robot devices in the industrial system, reducing both operational and management costs.

1 Introduction

In the recent decade, there has been an increasing demand from customers towards the manufacturing industry to provide more and more customized products. Personalized production is one of the key motivations for manufacturers to start leveraging new technologies that enable to increase, for instance, the flexibility of production lines. High flexibility, in general, is needed to realize cost-effective and customized production by supporting fast reconfiguration of production lines, as well as, easy application development. Fast reconfiguration and agile behavior can be achieved by moving the

---

Source code is available at https://github.com/slaki/nsdi22.

robot control from the pre-programmed local robot controllers to the cloud. In industrial robotics research, cloud robotics is a major topic and in the last years, several studies [9, 11, 13] have shown the benefits of connecting robots to a centralized processing entity: a) usage of more powerful computing resources in a centralized cloud especially for solving Machine Learning (ML) tasks; b) lower cost per robot as functionalities are moved to a central cloud; c) easy integration of external sensor data and easier collaboration or interaction with other robots and machinery; e) reliability of functions can be improved by running multiple instances as a hot standby in the cloud and the operation can immediately be taken over from faulty primary function without interruption.

Though centralized processing has clear benefits in making the management of industrial processes simple and flexible, cloud-based solutions cannot satisfy the low latency and high reliability network requirements of real-time industrial control (e.g., velocity or torque control of actuators, robot arms, conveyor belts, etc.). Industry 4.0 and 5G propose the use of edge computing infrastructure for this purpose, moving these tasks to the computing nodes located close to the industrial environment. Though the propagation delay can significantly be reduced with this setup, edge-computing nodes rely on the same virtualization technologies as remote cloud infrastructures. Existing solutions require real-time operating systems to eliminate the effects of CPU scheduling and ensure precise timing (e.g., in velocity control the velocity vectors need to be sent to the robot arms with accurate timing). Newer robot arms have 2 ms or less update time. The real-time velocity control of hundreds of such robot arms requires an ultra-fast response time that is hard to satisfy with traditional edge computing infrastructure.

With the advent of PISA switches [3] and the P4 language [2], a new era has begun in which programmable network devices can not only perform pure packet forwarding but simple computations as well. This trend led to the birth of a new computational paradigm called in-network computing, where server-based computations or a part of them are moved to programmable data planes. This new way of using network-

ing hardware can open up the fields for low-latency real-time calculations on the application level during the communication. Foremost, they can split long, distant control loops into smaller ones to deal with transport latency, enable computations at line rate and ensure real-time response time in orders of microseconds, solving the previously described problems of cloud and edge-cloud robotics.

In this paper, we investigate how cloud robotics can benefit from the advances of in-network computing. In particular, we propose a system in which high-level control of industrial processes can be deployed in the cloud (or edge cloud) while low-level speed control of the robot arms is offloaded to the programmable data plane (switch, smart NIC, or service card). Similarly to recent practical deployment options [6], we only assume reliable network connections with low latency between industrial robots and the programmable data plane. This design has the advantage that the high-level industrial controller does not require real-time OS and has less strict end-to-end delay requirements. Our vision is that P4-programmable data planes (e.g., smart NICs, service cards, switches) could complement the computational capabilities of cloud and edge cloud infrastructures for use cases where real-time operation, ultra-fast response time, high throughput, or all of these are required. Though the proposed method controls robot arms independently, we also demonstrate that it can easily synchronize the low level control processes of multiple robots and thus can potentially provide support for coordinated operation.

Moving low-level robot control to the network poses many challenges that are addressed in this paper: 1) How can velocity control be implemented with the limited instruction-set of programmable hardware data planes? 2) What is an efficient trajectory representation? 3) What to do if the entire trajectory does not fit into the memory? 4) How can match-action tables be used as playback buffers of trajectories? 5) How can trajectory segments be loaded in the limited memory of the switch and updated without violating timing requirements? 6) What constraints are needed for the data and control plane interactions? 7) How can the low-level control of multiple robot arms be synchronized? 8) How can switching to an alternative trajectory be solved in run-time (e.g., implementing a collision avoidance or emergency stop operations)?

2 Related Work

The related work of this paper covers a wide area of expertise from various research fields. We grouped them according to the different topics.

Traditional characteristics of robots. An industrial robot has many metrics and measurable characteristics, which will have a direct impact on the effectiveness of a robot during the execution of its tasks. The main measurable characteristics are repeatability and accuracy. In a nutshell, the repeatability of a robot might be defined as its ability to achieve repetition

of the same task. While, accuracy is the difference (i.e., the error) between the requested task and the realized task (i.e., the task actually achieved by the robot). For more details about the calculation of accuracy and repeatability, see [10]. The ultimate objective is to have both; a robot that can repeat its actions while hitting the target every time. When the current mass production assembly lines are designed, robots are deployed to repeat a limited set of tasks as accurately and as fast as possible to maximize productivity and minimize the number of faulty parts. The reprogramming of the robots rarely occurs, e.g., per week, per month basis and it takes a long time, e.g., days, requiring a lot of expertise.

Network aspects Authors of [7] compare the network protocols used nowadays in industry applications e.g., Modbus, Profinet, Ethercat. All investigated Industrial Ethernet (IE) systems show similar basic principles, which are solely implemented in different ways. Several solutions apply a shared memory and most systems require a master or a comparable management system, which controls the communication or has to be configured manually. Shared memory is implemented via data distribution mechanisms that are based on high frequency packet sending patterns. These packets have to be transmitted with strict delivery time and small jitter. IE protocols rely so heavily on the transport network that protocol mechanisms common in broadband usage like reliable transmission, error detection, etc. are not among the basic features of industrial protocols. Authors of [1] summarize the fundamental trade-offs in 5G considering various dimensions of block-lengths, spectral efficiency, latency, energy consumption, reliability, etc. Numerous aspects have to be solved during an industry automation task even when the robot stands still.

In-Network Industrial Control In-network control is a way to offload critical control tasks into network elements managed and organized through a remote environment. In the past few years, numerous papers offered solutions for In-Network Complex Event Processing (CEP). These works focus on sensor data-driven event triggering based on specific threshold values. Authors of [15] demonstrate such a system for a strongly delay-sensitive use case, controlling an inverted pendulum. By outsourcing the control to a distant controller, they show how a very low RTT of 5-20ms can break the entire system or make it oscillate badly. By combining in-network processing with the distant controller, they were able to utilize the ultra-low latency of local communication, and the control of the pendulum showed identical results as with fully local control. This paper mainly focuses on the implementation of the LQR controller in P4 and the limitations of the P4 language. Though the method we propose in this paper also uses a controller (PID-like) in the middle of the pipeline, it goes much further by providing an abstract representation of function components with error bounds that can potentially be used in any controller algorithms. In addition, our approach also handles many other problems: trajectory-based

control, switching between trajectories, synchronization of multiple robots, etc. In [12] authors demonstrate their own P4-based CEP rule specification language. P4CEP's system model works with a collection of end-systems that are interconnected by programmable network processing elements. End-systems are differentiated into event sources, and event sinks where the sinks can react to certain conditions observed by the event sources. FastReact [20] is another In-Network CEP system that advocates the idea to outsource parts of an industrial controller logic to the data plane by making the programmable switches able to cache the history of sensor values in custom data structures, and trigger local control actions from the data plane. [4] shows a robot control system where a P4 switch is located between an emulated robot arm and the controller. The switch can analyze both sides of the traffic. If it detects that a position threshold is violated by the robot, it sends back an emergency stop message within a very short time due to the local communication. This work only covers this simple failure detection scenario and cannot deal with the more advanced control of robot arms we show in this paper.

3 System Design

The main goal of this paper is to demonstrate the feasibility and practical benefits of programmable data planes in low-level industrial control. To this end, we show how real-time velocity control of robot arms can be implemented in P4-programmable network devices and how they can be integrated into the existing industrial ecosystem. Fig. 1 depicts the high-level architecture of the proposed system, enclosing one or more robot arms, a P4-switch, and an industrial controller. It is important to note that this is a practical deployment option. The first phase of the introduction of wireless communication into production cells looks similar [6].

Robot arms. We assume simple robot arms without in-built intelligence. Each robot arm consists of a number of joints controlled by actuators (i.e., servo motors). The actuators work independently, stream their internal state (position and velocity) at a constant frequency (generally in the range of 100Hz-1kHz) and require velocity control messages at a predefined rate (generally 100Hz-500Hz) to keep the movement smooth. Note that lost command messages may cause lags in the movement or deviance from the desired path to be followed. In our system model, each robot arm is handled as a set of actuators controlled in sync. However, many complex industrial processes also require the synchronized operation of multiple robots (or other devices like conveyor belts, etc.). In the proposed system, this case can naturally be deduced to the single robot case by handling the cooperative robots as a single entity with all the actuators of the participating individual robots.

P4-switch. A programmable packet processing device supporting the P4 language [2] (e.g., PISA switch, smartNIC, or distributed service card) that processes the status streams of
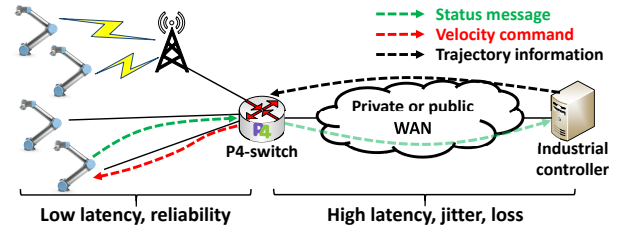

Figure 1: System overview.

the robot joints and generate the velocity control commands from the state messages and the desired trajectory provided by the industrial controller. We assume a highly reliable network connection with suitably small propagation delay (depending on the robot's control frequency) between the P4-switch and the robots.

Industrial controller. It is responsible for coordinating the industrial processes at a high-level and thus planning the trajectories to be followed by the robot arms, re-planning trajectories if needed (e.g., for collision avoidance), verification of the process, failure detection and response, and synchronizing high-level processes. In our system design, the controller could be deployed at remote or edge cloud infrastructure. In the case of remote cloud deployment, the delay between the switch and the industrial controller could be in the order of 10-100ms with significant jitter. In both cases, the high-level industrial controller does not require real-time OS and thus can operate in a VM. Note that the industrial controller also gets the status information of the robots needed for tracking the whole industrial process, but cannot directly send commands to the actuators. Instead, it fills the match-action tables of the switch with a sequence of trajectory points needed for the P4-switch for controlling the robots at a low-level.

During operation, each robot arm executes the trajectory planned by the industrial controller. A trajectory is represented by a sequence of trajectory points (TPs), where each TP has a unique identifier and is associated with a relative timestamp (starting with 0) and the expected state (joint velocities and positions) of the robot arm at the given point of the operational timeline. Two consecutive TPs may be far from each other in both time and joint spaces. In the proposed method, the P4-programmable switch is responsible for the transition between the two TPs by continuously updating the joint velocities of the robot arm.

A trajectory example is depicted in Fig. 2. The initial trajectory plan on the top is a sequence of snapshots describing the robot states at discrete points of time. In the snapshot images, the orange arm illustrates the final configuration to be reached and the other denotes the desired state in the given TP. A robot state is described by two vectors representing the desired joint velocities and joint positions. Note that though the robot arm moves in the Cartesian space (as shown in the figure), the industrial controller maps the trajectory to the joint space (with
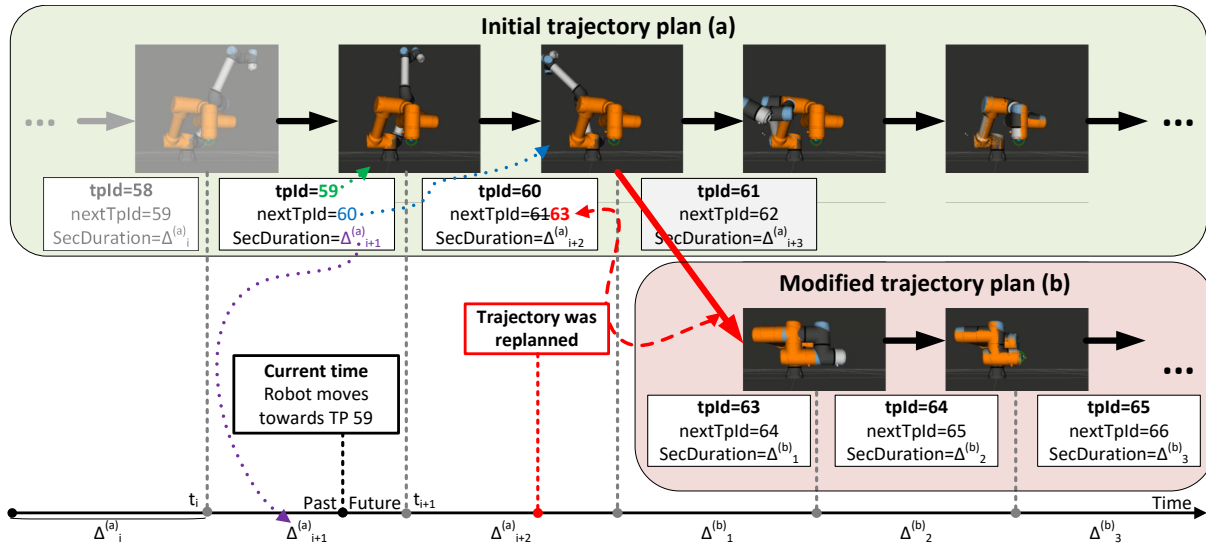
Figure 2: Trajectory example with re-planning.

units of rad/s and rad). One can also see that the transition from one TP to another needs to be performed in the allocated section duration of Δ_j. In the figure, the robot is heading TP 59. The new joint velocities to be set are calculated from the current state of the robot arm and the desired state in TP 59. Older TPs (e.g., 58 in the figure) have become obsolete. As soon as the target TP is reached, we switch to the next TP (60 in the example), heading the new associated robot state and also considering $\Delta_{i+2}^{(a)}$ dedicated for the transition (from TP 59 to 60).

3.1 System Requirements

In this section, we identify the minimal set of requirements needed for low-level real-time control of robot arms in most industrial use cases. We need to consider them during the implementation of the proposed system.

Velocity-control requirement. The smallest building blocks to be controlled are the actuators in our system. Actuators can be controlled independently. Each of them periodically generates status messages carrying the current joint velocity (rad/s) and joint position (rad) values. These messages first need to be parsed by the P4-switch responsible for low level control. Then, the switch has to calculate the new velocity value by applying feed-forward control (e.g., PID) that combines the state, timing, and trajectory information. Finally, the result shall be written into a command message and sent back to the actuator. Actuators are using different state reporting and command execution frequencies (generally the former is higher). Actuators operate at a given frequency. Each actuator first waits for a command message in a time window of constant length. If the time window is over, the actuator executes the command. If multiple commands are received in a time

window, only the latest is kept and all the others are dropped.

Timing requirement. The precise timing of control commands is crucial since the actuators of the robot arms expect incoming commands with a given frequency and do not tolerate large timeouts and jitter. In case of bursty arrival, a part of the commands may not be executed, leading to unexpected deviations from the desired trajectory. For example, an UR5e robot arm expects commands at 125 Hz, requiring a command message every 8ms. In addition, there are timing requirements between the P4-switch and the industrial controller on loading trajectory information. This requirement especially important if the entire trajectory cannot be stored in the switch (or it is not intended), and the controller periodically loads new TPs and deletes obsolete ones.

Synchronization requirement. Though we assume that actuators can be controlled separately, they are not independent. They belong to a single physical structure with its own kinematics. Thus, the actuators of a single robot need to be coupled in the control process. In addition, most industrial processes require the cooperation of multiple robot arms. Synchronization requirements can be defined on different time-scales. For example, if a robot stops in a position and then another process is started, but there is no strict time constraint (e.g., few seconds are acceptable) between the two processes, a remote industrial controller can even solve the synchronization. However, in several cases, this light synchronization is not enough, and thus the low-level control processes also need to work in sync (on a millisecond or sub-millisecond scale).

Trajectory switching requirement. The industrial controller continuously monitors the whole industrial process, and intervenes if needed (e.g., in case of failure or simple reconfiguration, or for collision avoidance purposes). This case is illustrated by Fig. 2 where the trajectory is modified (the red

point on the timeline), resulting in that after TP 60 the robot moves towards TP 63 instead of 61. Trajectory switching is needed when a robot is reconfigured or when an obstacle appears in the robot cell and collision avoidance can be ensured by the new trajectory.

Communication requirement. To reduce packet processing overhead in the P4-switch, we assume that robot arms apply a datagram-based communication protocol (e.g., native Ethernet frames, UDP packets, etc.) for sending status information and receiving commands. Both status and command messages consist of simple decimal fields in a binary format that can be parsed by the P4-switch with ease.

4 Robot Arm & Network Protocol

The robot arm used in our experiments is an UR5e [17]. UR5e is a lightweight, adaptable collaborative industrial robot with six joints (6-DOF). It is commonly used in research as it has a programmable interface, can be remotely controlled, provides industrial grade precision, and can operate alongside humans with no safeguarding. The robot vendor also provides a real-time emulation environment (URSim) that is fully compliant with the real robot arms and thus can be used for testing validation purposes.

UR5e can receive external commands described in UrScript [19] language via its network interface. It communicates with external controllers over TCP by default. However, the network protocol can be customized by adding a URCap [18] plugin (called daemon) to the robot. To make the communication simple and stateless, we created a URCap daemon implementing the translation between the original TCP-based and our UDP-based protocols.

During the protocol design we considered two practical aspects: 1) P4 capable devices are not suited for deep packet inspection, and thus cannot parse the entire content of large packets. It implies that every important field used for robot control has to be close enough to the beginning of the packet. 2) Both status and command messages of the original communication interface rely on floating point fields. However, the P4 language does not support floating-point arithmetic. This problem can be handled by multiplying each floating-point value with a properly large constant and then using the standard decimal operations. Though it is possible to implement this conversion in P4, it is much simpler and comfortable if the value is already in a decimal format in the used protocol.

Considering the above aspects, we use the same header structure for status and command messages encapsulated into simple IP/UDP packets. The introduced robot header (*rh*) consists of four fields: 1) a **robot ID** (*rh.RId*) used as a unique identifier of the robot arm, 2) a **joint ID** (*rh.JointId*) which determines the joint (or in general the actuator) of the given robot, 3) a **joint velocity** (*rh.velocity*) expressing the current speed (in *rad*/s) of the given joint in the status messages or the new joint-speed value to be set in the commands, and 4) a

joint position (*rh.position*) which is the current position (in *rad*) of the given joint in the status messages, and unset in the commands.

5 Velocity Control in Data Plane

Though our prototype is implemented in P4-16 with the Tofino Native Architecture (TNA), we aim at keeping the data plane description in this section general. In our model, the switch consists of two packet processing pipelines: an ingress and an egress. The two parts have different roles and responsibilities in the proposed implementation:

- **Ingress pipeline:** This part is responsible for 1) determining the current TP for the robot arm the status packet is sent by, 2) stepping the current TP to the next TP along the trajectory if required, or 3) switching to another trajectory in case of re-planning.

- **Egress pipeline:** This block solves the low-level velocity control by calculating the new joint velocity value based on the available information (state packet and trajectory).

5.1 Ingress pipeline

We assume that each TP can be identified by a unique ID. The memory layout of the ingress pipeline is depicted in Fig. 3. One can observe that we maintain three registers for each robot to be controlled. They store the identifiers of the current (REG_{tp}), the next (REG_{nextTp}) TPs, and the absolute timestamp ($REG_{nextTime}$) when the control has to step along the trajectory to the next TP. Fig. 2 provides a good illustration of the role of these three values. Accordingly, the robot moves towards the current TP (59) which should be reached at t_{i+1} ($REG_{nextTime}$) when we step forward to the next TP (60).

The ingress pipeline also contains two tables for storing the trajectory as a sequence of TPs and branching points where we can switch to another trajectory. Table `TPStepper` represents the trajectory to be followed by a robot arm as a linked list of TP identifiers. For each TP p, it stores the duration needed for moving from the previous TP to p and the identifier of the next TP that follows p along the trajectory. One can observe that the next TP determines how the robot arm continues its operation after reaching p.

The current and next TPs usually belong to the same trajectory, but in some cases, re-planning is required. Table `TrajectorySwitcher` solves this problem by switching between two trajectories. If there is a TP p along the original trajectory which could also be the starting point of the new trajectory, the switch can be implemented by replacing the next TP of p with the appropriate TP along the new trajectory. Thus after the branching point p, the robot arm starts following the new trajectory also loaded into table `TPStepper`.

Let us consider the example in Fig. 3 (see Fig. 2 for illustration). Table `TPStepper` is applied at time t_i when TP 59 becomes the new TP ($m.tpId = 59$) the robot arm is heading towards. At this point of time, the next TP is unknown and is filled from the table. The table also provides the section duration ($m.secDuration = \Delta_{i+1}^{(a)}$) allocated for reaching TP 59. This information is used for determining the absolute timestamp ($t_i + \Delta_{i+1}^{(a)}$) when the current TP is replaced by the next one (TP 60) and then table `TPStepper` is applied again. Though it sets REG_{nextTp} to 61 (next TP along the initial trajectory), table `TrajectorySwitcher` overwrites it with 63, the starting point of the new trajectory.

Algorithm 1 describes the ingress pipeline at a high abstraction level. At arrival, the status message from a robot executes the program block starting with line 3. First, the trajectory state ($TpId$, $nextTpId$ and the $nextTime$) is red from the registers. $TpId$ denotes the current TP the robot is currently heading towards and $nextTpId$ identifies the next TP. Then table `TrajectorySwitcher` is applied that replaces the $nextTpId$ if the current TP is a branching point. In most cases, there is no hit in this table. In line 6, we check if $nextTime$ is reached. If this condition is true, further actions (see line 11-16) are needed since we have to move to the next TP, update states (table `TPStepper`) and write them into the registers. In high-performance hardware data planes like Barefoot Tofino, registers can only be accessed once during the pipeline to ensure line-rate performance even at the Tbps scale. This constraint can be resolved by resubmitting the packet (lines 8-9). In this case, the ingress pipeline is executed twice only. Though packet resubmission can reduce the overall throughput, in practice this step is only performed when the current TP is reached. Note that in software targets the proposed pipeline could be implemented without the need for resubmission, but in turn, we can expect higher latency and performance limitations.

The proposed implementation has further practical benefits. In case of repetitive tasks which is usual in industrial scenarios, we can simply create loops in table `TPStepper` by setting the next TP to a TP visited previously. The synchronization of different robot arms can be solved either by merging the multiple robot arms into a single entity whose TPs represent the joint states of all participating robots or by creating a self-loop at the starting point of trajectories to be synchronized. In the latter case, if the section duration is long enough for inserting branching points to trajectories to be executed into table `TrajecotrySwitcher`, the internal clock of the P4-switch ensures that robot arms start operating at the same time and are kept in sync during the industrial process.

5.2 Egress pipeline

The egress pipeline is responsible for calculating the velocity value to be set from the current state of the robot joint and the current TP ($tpId$). The new velocity value is computed

---

Algorithm 1: Ingress pipeline (pseudo-code)

Robot header: rh, **Metadata:** m;
Registers: REG_{tp}, REG_{nextTp}, $REG_{nextTime}$;
Tables: TrajectorySwitcher, TPStepper;
apply block

1   **if** rh.*isValid()* **then**
2   **if** m.*resubmitted==0* **then**
3   m.tpId = REG_{tp}(rh.RId);
4   m.nextTpId = REG_{nextTp}(rh.RId);
5   m.nextTime = $REG_{nextTime}$(rh.RId);
6   TrajectorySwitcher.*apply()*;
7   **if** m.nextTime>*now()* **then**
8   m.*resubmit_needed* = 1;
9   m.*resubmit_data* = m.nextTpId;
10   **else**
11   m.tpId = m.*resubmit_data*;
12   TPStepper.*apply()*;
13   REG_{tp}(rh.RId) = m.tpId;
14   REG_{nextTp}(rh.RId) = m.nextTpId;
15        $REG_{nextTime}$(rh.RId) += m.secDuration;
16   send_back();
17   **else**
18   *Handling normal traffic (e.g., l2 forwarding)*;

---

by a simple PID-like controller, as the weighted sum of three values:

$$v_{new} = v_{curr} + c_1(v_{trg} - v_{curr}) + c_2(p_{trg} - p_{curr}),$$

where c_is are constants, v_{curr} and p_{curr} denote the current speed and position of the robot joint while v_{trg} and p_{trg} are the desired joint velocity and position in the current TP. One can observe that the new velocity can be composed of three linear transformations: $(1 - c_1)v_{curr}$, $c_1 v_{trg}$, $c_2 p_{diff}$, where $p_{diff} = p_{trg} - p_{curr}$. Each actuator may have different physical properties and thus require different c_i constants in the transformations. The three `Transform` tables in Fig. 4 are used for approximating these linear transformations.

The egress control block is described in Algorithm 2. We first apply table `TargetData` to obtain the desired joint speed and joint position in the current TP ($m.tpId$). The actual state of the robot joint is carried by the robot header (rh). Lines 3-7 perform the primitive calculations needed for the P-controller mentioned previously. The new velocity is calculated as a sum of different components. Each component is calculated from metadata fields (diffPos stores the position difference) filled previously or from header fields by a transformation. The transformations are approximated by ternary or longest-prefix match (LPM) tables filled in run-time (see Sec. 5.3). If the calculated velocity value is too large, it can cause damage to the robot arm. To take the physical limits of the robot joints into account we introduce the table `LimitVelocity` checking
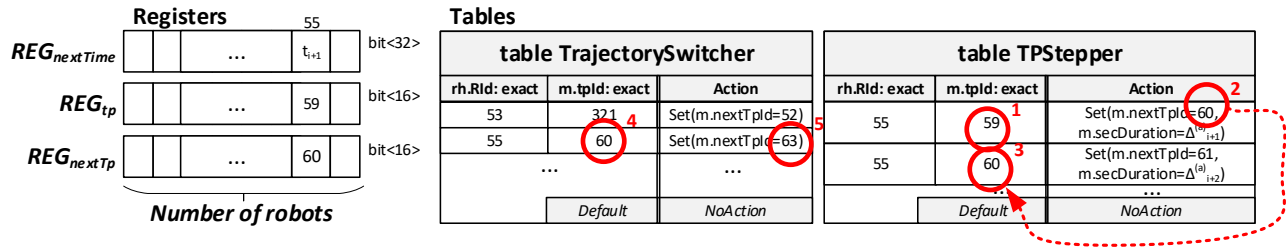

Figure 3: Memory layout at ingress.

whether the calculated velocity value is outside of the safety range, and mapping it into the normal range if needed. The calculated joint speed is encoded into the velocity field of robot header *rh* and sent back to the robot as a command message (lines 9-11).

Algorithm 2: Egress pipeline (pseudo-code)

Robot header: rh, **Metadata:** m;
Registers: -;
Tables: TargetData, LimitVelocity,
    TransformTrgVelocity, TransformCurrVelocity,
    TransformDiffPosition;
apply block
1   **if** rh.*isValid()* **then**
2   TargetData.*apply()*;
3   m.diffPos = m.trgPos - rh.position;
4   TransformTrgVelocity.*apply()*;
5   TransformCurrVelocity.*apply()*;
6   TransformDiffPosition.*apply()*;
7   rh.velocity += m.trgVel + m.diffPos;
8   LimitVelocity.*apply()*;
9   swap_ipAddresses();
10   swap_udpPorts();
11   clear_checksums();
12   **else**
13   *Handling normal traffic*;

5.3   Approximating transformations

The new velocity value is calculated by applying transformations on some header or metadata fields. In our proof-of-concept P-controller, these transformations are simple multiplications with predefined constants, but this design enables us to apply even non-linear mappings.

Such a transformation can be approximated by a Longest-Prefix-Match (LPM) or a ternary-match table as depicted in Fig. 4. The match key is the parameter of the function (e.g., a header or metadata field), considering the most significant n bits starting with 1 (positive case) or 0 (negative case), as illustrated in Fig. 5. The action parameter is the function value

calculated from the significant bits only. Since we only use simple weight functions in our implementation, the relative error of the approximated output equals the relative error of the input, more precisely the relative error of the estimation based on the most significant n bytes. The estimated value for the input can vary between the largest and smallest possible values with the given prefix. During this process, we skip the leading zeros (or ones in case of negative values) and ignore the last k bits. Depending on this estimation, the relative error is less than or equal to $1/2^{n-1}$.

This approach fits well with the velocity control use case. If the input is small – suggesting that we are close to the target TP, we need to make a more precise movement – the approximation has a small absolute error. If the input has a higher absolute value – meaning that we are far from the target value, and high precision control is not needed – the method provides an acceptable higher absolute error.

This method can be improved with a small trick. The number of possible outputs is exactly the number of ternary entries. However, we can calculate the approximated value of $(c-1)x$ instead of cx and add one more x to the result in the P4 program. This technique applied in Table `TransformDiffPosition` helped to improve the stability of the applied P-controller.

5.4   Limiting joint velocities

The different joints have their own physical properties that determine the maximum applicable velocity. To check the speed constraints and limit the velocity if needed, we apply table `LimitVelocity`. Let x be the velocity value (*rh.velocity*) to be tested and c be a constant value. Starting with the positive case, we can always decide whether $x > c$ if x has the same n long prefix as c but $x[n+1] = 1$ and $c[n+1] = 0$. Note that $x[1]$ denotes the most significant bit of x. The negative case is similar. If x has the same n long prefix as c but $x[n+1] = 0$ and $c[n+1] = 1$ then $x < c$. For an input of k bits, we need at most k entries in the table for each constant check. Fig. 4 shows a small example with the necessary prefix checks, considering a 16-bit long input value and predefined constant c. In the case of signed inputs, the first bit shall be handled carefully, but comparing to a negative number can be done similarly.
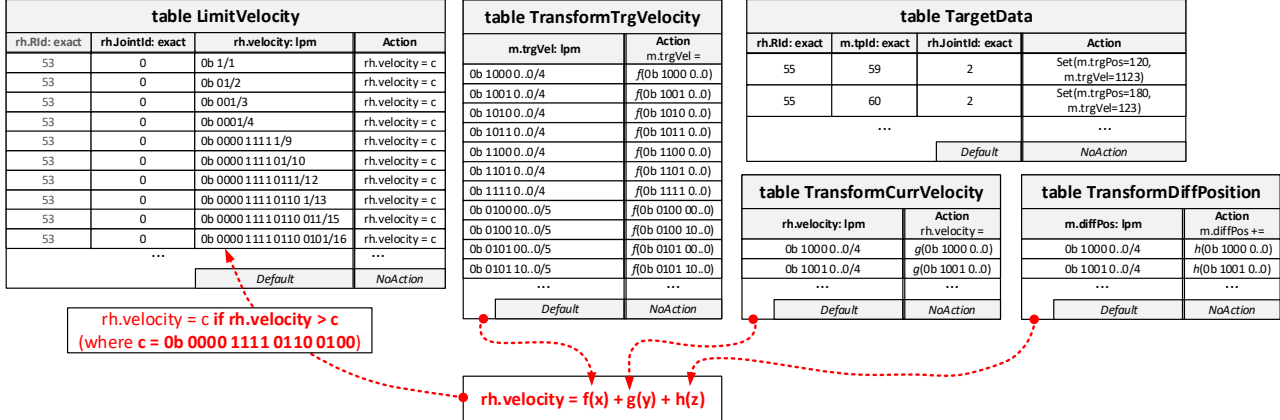
Figure 4 tables

table LimitVelocity

rh.RId: exact	rh.JointId: exact	rh.velocity: lpm	Action
53	0	0b 1/1	rh.velocity = c
53	0	0b 01/2	rh.velocity = c
53	0	0b 001/3	rh.velocity = c
53	0	0b 0001/4	rh.velocity = c
53	0	0b 0000 1111 1/9	rh.velocity = c
53	0	0b 0000 1111 01/10	rh.velocity = c
53	0	0b 0000 1111 0111/12	rh.velocity = c
53	0	0b 0000 1111 0110 1/13	rh.velocity = c
53	0	0b 0000 1111 0110 011/15	rh.velocity = c
53	0	0b 0000 1111 0110 0101/16	rh.velocity = c
...			...
		Default	NoAction

rh.velocity = c **if rh.velocity > c**
(where c = 0b 0000 1111 0110 0100)

table TransformTrgVelocity

m.trgVel: lpm	Action m.trgVel =
0b 1000 0..0/4	f(0b 1000 0..0)
0b 1001 0..0/4	f(0b 1001 0..0)
0b 1010 0..0/4	f(0b 1010 0..0)
0b 1011 0..0/4	f(0b 1011 0..0)
0b 1100 0..0/4	f(0b 1100 0..0)
0b 1101 0..0/4	f(0b 1101 0..0)
0b 1111 0..0/4	f(0b 1111 0..0)
0b 0100 0..0/5	f(0b 0100 00..0)
0b 0100 10..0/5	f(0b 0100 10..0)
0b 0101 00..0/5	f(0b 0101 00..0)
0b 0101 10..0/5	f(0b 0101 10..0)
...	...
Default	NoAction

table TargetData

rh.RId: exact	m.tpId: exact	rh.JointId: exact	Action
55	59	2	Set(m.trgPos=120, m.trgVel=1123)
55	60	2	Set(m.trgPos=180, m.trgVel=123)
...	...	...	...
		Default	NoAction

table TransformCurrVelocity

rh.velocity: lpm	Action rh.velocity =
0b 1000 0..0/4	g(0b 1000 0..0)
0b 1001 0..0/4	g(0b 1001 0..0)
...	...
Default	NoAction

table TransformDiffPosition

m.diffPos: lpm	Action m.diffPos +=
0b 1000 0..0/4	h(0b 1000 0..0)
0b 1001 0..0/4	h(0b 1001 0..0)
...	...
Default	NoAction

rh.velocity = $f(x) + g(y) + h(z)$

Figure 4: Memory layout at egress.

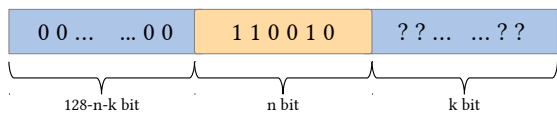| 0 0 ... ... 0 0	1 1 0 0 1 0	? ? ... ... ? ?
128-n-k bit	n bit	k bit

Figure 5: Considering the most significant n bits starting with 1 (positive case).

6 ROS integration

In this section, we briefly introduce our industrial controller implementation based on the Robot Operating System (ROS) [14] and its MoveIt [5] library used for generating and executing trajectories in a robot agnostic manner. ROS is an open-source robotics framework used in various robotics-related research since it can easily be extended and customized for specific use cases. Our ROS-based industrial controller uses MoveIt for motion planning and mobile manipulation of robots. In the proposed system, it generates a *JointTrajectory* message containing an array of points (timestamps, 6 joint positions, 6 joint velocities), as UR5e has six joints (as shown in Fig. 2).

The architecture of the industrial controller is shown in Fig. 6. The components developed to support the proposed system are marked by gray. They have been integrated with the standard MoveIt architecture consisting of trajectory generation using MoveIt planning, trajectory execution with MoveIt using standard ROS interface, and communication via the ROS driver of the UR5e arm.

To generate trajectories we can use RVIZ, a ROS visualizer software, with a MoveIt Motion Planning Plugin. Using RVIZ, we can generate trajectories interactively from a start point to a selected endpoint. Another way to generate trajectories is by 1) creating waypoints in Cartesian space, 2) then sending those points to a ROS node, 3) it computes a trajectory in joint space (defined by the joint angles of the robot) and 4)
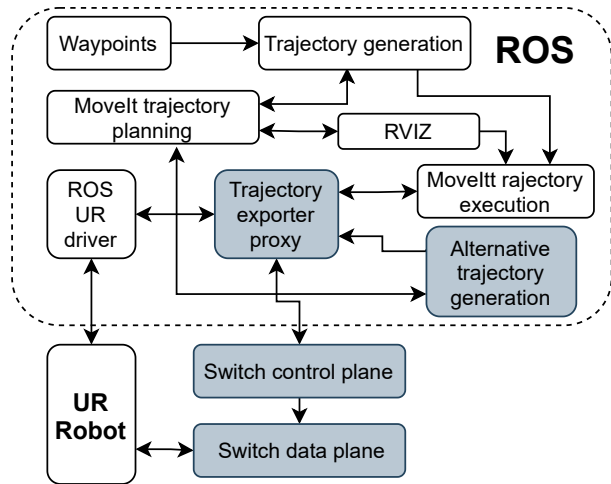
Figure 6: Trajectory generation and execution

finally it visits them in order.

The resulting joint trajectory is sent to MoveIt trajectory execution, which uses the ROS action interface defined by the UR driver to execute the trajectory on URSim/UR5e or it can also send the trajectory to the P4-switch's control plane via the Trajectory Exporter proxy, which fills the appropriate tables and let the switch execute the trajectory instead of the ROS UR driver.

6.1 Alternate trajectory generation

We developed Alternate trajectory generation to extend the existing capabilities of the system. Alternate trajectory generation and execution is a feature that leverages the ability of the P4 system to quickly change chains of trajectories, to execute prepared alternate trajectories in response to external changes e.g., in the robot's surroundings.

The alternate trajectory generation node uses MoveIt's tra-

jectory planning to prepare multiple branching trajectory fragments, then concatenates them into a single trajectory. The alternate trajectories are placed after each other, therefore the timestamps of the whole branching trajectory are not strictly incremental. Fig. 7 shows this process, the numbers indicate the timestamps of the trajectory's start and endpoints.
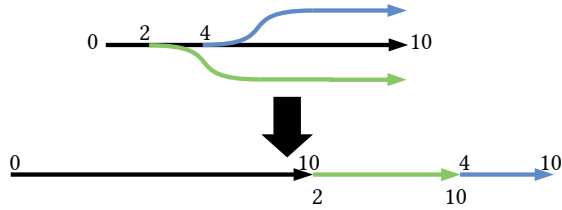

Figure 7: Generating a single trajectory from alternate ones

As the timestamps do not strictly increase, we can not use the trajectory execution of MoveIt. MoveIt is not able to execute alternate trajectories. Therefore we send the encoded trajectory to a proxy ROS component (node) which forwards the trajectory to the P4-switch.

We are also able to generate alternate trajectories on the fly when a trajectory is already being executed. We achieve this by keeping track of what is the current time in the currently executed trajectory. As we know the current time, we know where the robot will be in a Δt time. That point can be the start point of an alternate trajectory. To ensure a smooth transition during the switch between alternate trajectories we need to estimate the position of the switching as accurately as possible. To do this we need to estimate the 1) the latency between the industrial controller and the switch (RTT), and 2) the expected trajectory generation time ($t_{processing}$). For the estimation of the start position ($p_{start}(t)$) at time (t) we came up with the following formulae:

$$\Delta t = t_{processing} + RTT$$
$$p_{predict}(t) = p_{traj}(t + \Delta t)$$
$$+ p_{status}(t) - p_{traj}(t)$$
$$+ (v_{status}(t) - v_{traj}(t)) \times \Delta t$$
$$p_{start}(t) = \lfloor p_{predict}(t) \times \frac{1}{g(t)} \rfloor * g(t)$$

Where the error is estimated on the trajectory calculation side by calculating the difference of the position of the joints received in the last status message and the executed trajectory position. This is further adjusted by the difference of the current velocity of the joints and trajectory velocity times Δt. A binning of the values with an integer division and multiplication with the original granularity ($g(t)$) is applied on the predicted position value to replicate the behavior of the ternary table on the trajectory planner side and consider the granularity of the number representation in the specific time. $g(t)$ can be derived from the maximum of relative error (M_{rel}, see Sec. 5.3) by $g(t) = lpm(p_{traj}(t), M_{rel})$.

Fig. 8 shows an example on the error of $p_{start}(t)$ compared to $p_{status}(t + \Delta t)$, which is the joint position at the time of switching.
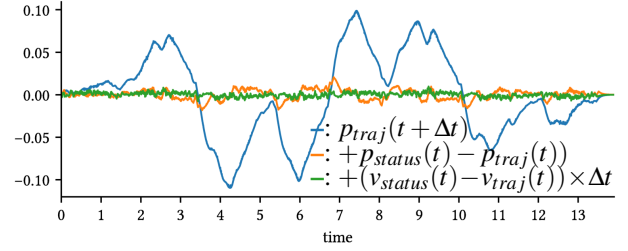

Figure 8: An example on the error of $p_{start}(t)$ compared to the later joint state

7 Evaluation

We carried out several experiments analyzing whether the proposed implementation can hold the identified system requirements. To this end, we deployed a simple testbed consisting of two servers (AMD Ryzen Threadripper 1900X 8C/16T 3.8 GHz, 128 GB RAM) and a Barefoot Tofino-based switch (STORDIS BF2556X-1T). One of the servers (called Server-A) is equipped with a dual port 10 Gbps NIC (Intel 82599ES) that supports hardware-based timestamping. This node is connected to the switch via two 10 Gbps links. The other server (Server-B) is equipped with a Mellanox ConnectX-5 dual port NIC whose ports are connected to the switch via two 100 Gbps links. For latency experiments, we used MoonGen tool [8] on Server-A with hardware-based timestamping to generate robot and mixed traffic. During the latency measurements, Server-B continuously generated non-robot background traffic with IP/TCP packets of size 1280B. We aimed to demonstrate the case that some ports of the P4-switch are dedicated to handling robot traffic while others forward normal traffic in parallel. For operational experiments, we used a real UR5e robot arm connected to the switch and Server-B was running one emulated UR5e [17] robot using the official URSim robot emulator. Note that the emulator provided by the robot vendor is fully realistic and works in real-time. During the experiments, we did not realize notable differences between the emulated and the real robot arm. In this scenario, our ROS-based industrial controller was run on Server-A and communicated with the control plane of the switch, loading and removing trajectory points.

The performed evaluation scenarios were designed to assess the proposed system in various common robotic use cases: 1) Pick and place actions: See Sec. 7.3, 2) Welding, painting, gluing: See Sec. 7.4, 3) Robot to robot collaboration: See Sec. 7.2, 4) Heterogeneous sensor and actuator deployment in the robot cell: See Sec. 7.1, 5) Agile control, safety, robot to human collaboration: See Sec. 7.5. The estimation

about scalability is covered by Appx. A.

7.1 Response time analysis and traffic load

A robot cell usually contains various sensor and actuator elements, meaning that there are other traffic sources in the robot cell than the one generated by the robot arm itself. The Supervisory Control And Data Acquisition (SCADA) systems also generate considerate background traffic. This is why it is important to evaluate the proposed system on a heterogeneous traffic mix. To this end, we carried out latency measurements under various traffic loads. In this scenario, MoonGen (Server-A) sent latency probes in every ms, mixed with various background traffic. The latency probes were valid robot status messages and thus they went through the entire robot control pipeline.

We evaluated the system with two different background traffic: **1) Simple IP packets of 64B and 1280B sizes** were generated at variable sending rates (1-10 Gbps). The switch applied simple port forwarding and only the latency probes went through the robot-control pipeline. With small packet sizes the observed response time of robot traffic was in the range of $[0.6\mu s, 1.3\mu s]$. With packet size of 1280B the response time shifted towards $2\mu s$ as the load increased. This phenomenon was caused by one or more large background packets wedging between two latency probes. Note that at 10 Gbps transmitting a packet of size 1280B takes approx. $1\mu s$. We also compared these measurements to the latency of a simple port forwarding program, the differences were not significant ($<0.2\mu s$). **2) Robot status messages** were generated as background traffic, and thus all the packets went through the entire robot control pipeline. The latency results were basically identical with the previously described case of using 64B IP packets. The response time was ranging between $0.6\mu s$ and $1.3\mu s$.

Though these measurements only show the response time in under-loaded situations without queueing effect, they are represented in most industrial environments where a number of assumptions can be made: **1) predictable and stable load** since the device settings determine the packet generation frequencies; each device operates as a constant bit-rate source. The packet sizes are known and thus the overall load can easily be predicted. For example, a 6-DoF robot operating at 500 Hz (e.g., UR5e sends status messages at this rate; sending in every 2ms) generates approx. 1.5 Mbps status traffic on the upstream direction. Thus, the packet processing pipeline is required to ensure non-blocking operation at 3000 packets/s for a single robot. Considering 1000 robot arms which is far above the number of robots used in industrial setups nowadays, the required forwarding rate is 3M packets/s (approx. 1.5 Gbps) on average. However, considering synchronized robots whose status messages are sent within a short time window, the bursty arrival at the P4-switch can lead to higher peak rates to be handled. For example, if status

messages from all the robots arrive within a time window of 1ms (50% of the 2ms sending interval), the observed temporal rate could be 3 Gbps or higher. One can observe that these arrival rates can easily be served by currently available P4-hardware including both smartNICs, DSCs, and P4-switches. **2) The robot-control traffic can be separated** from other traffic either by assigning dedicated ports and/or pipes to robot traffic or using simple priority queues giving higher priority to industrial traffic than background packets. Note that priority queueing is supported by most of the networking elements (also including non-P4-programmable ones). This scenario is examined in more detail in Appx. C.

We also tested our pipeline enforcing the packet resubmission at ingress, but it had no visible effect on the latency distribution. Finally, we repeated all the delay measurements with generating robot traffic at 100 Gbps from Server-B, but it has no effect on the observed latency at Server-A.

7.2 Synchronization measurements

Robot to robot collaboration is an important use case in any industrial robot cell deployment. To speed up the assembly process a usual deployment contains a robot arm moving the part to be worked with into various reachable positions for the other arm that has various grippers and executes a specific assembling order. The two arms need perfect synchronization otherwise the resulting product is faulty.

In this operational experiment, we launch the real UR5e robot arm and an instance of the URSim robot emulator, both are controlled by the switch and we start the trajectories in sync and out of sync. Fig. 9 shows the time shift between the start times of the two robots. The experiment was repeated 20 times. In the synchronized case, we created a single entity from the two robot arms with 2×6 joints and launched the trajectory by adding an entry to the TrajectorySwitcher table. The result is a fully synchronized operation as depicted by blue in the figure. In the non-synchronized case, the two entries are inserted independently to start the two robots. Note that we observe an 8 ms time shift in the worst case that is comparable with the control frequency of the robots (125 Hz) and can simply be caused by the 8 ms real-time window of the robots. Though the observed time-shift is basically negligible for two robot arms, we assume that it may be much more significant if a larger number of robots is launched independently.

7.3 Accuracy at stop position

The accuracy and repeatability of a robotic arm are essential key performance indicators (KPIs) that need to be maintained even in a cyber-physical-system, i.e., remote control over the network. The basic pick and place, and palletizing use cases mostly depend on them. It is a bare minimum that the proposed system works well in these use cases.
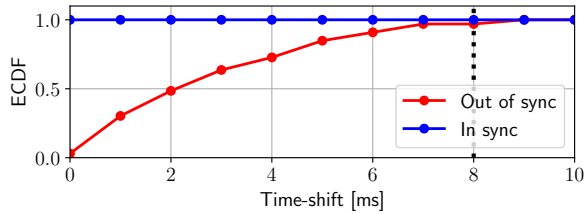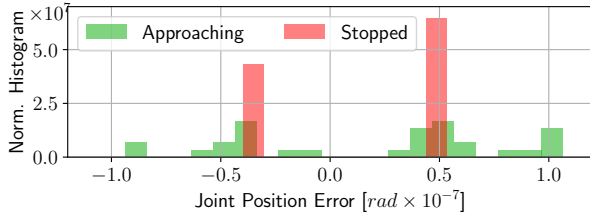
Figure 9: Time shift.

Figure 10: Joint position error at stop.

We performed experiments to analyze the accuracy of the control at the stop position (at the end of the trajectory) of the robot arm. One can observe in Fig. 10 that the error from the expected joint position was about 0.5×10^{-7} *rad* which was caused by the applied number representation (see Sec. 5.3). The green bars illustrate the deviance when the joint speed is not absolute zero, but the joint is close to its target position. Note that 0.5×10^{-7} *rad* error in the joint position corresponds to $0.5\mu m$ with a 1m long robot arm. In a robot arm with multiple joints, the cumulative position error is still in the order of micrometers.

7.4 Accuracy along the trajectory

The assembling quality and the endurance of a product mostly depend on the quality of gluing, welding and painting work. To ensure this, the robot needs to be accurate not only in the goal positions but all along the planned trajectory.

Though the proposed implementation is highly configurable and supports the fine tuning of the applied P-controller, more advanced controllers (e.g., PID) can obviously provide more precise control. In this experiment, we measure the accuracy of the robot head at the trajectory points and compare the results to the PID-based velocity control of ROS. Both ROS and the emulated robot run on the same server, representing ideal circumstances for ROS-based control.

Fig. 11a and 11b depict the TPs (green points) as well as the path of the tool at the end of the robot arm (solid curves) for controls based on ROS and P4-Switch, resp. Both paths show a similar character. The accuracy of the two solutions is presented in Fig. 11c. ROS's fine-tuned PID-controller provides a 0.1 mm accuracy in the worst case which is 3.2

mm in the case of our proof-of-concept P-controller. The median accuracy values are 0.04 mm and 2.23 mm for ROS and P4-Switch, respectively.

7.5 Continuous table management

Industry 4.0 introduces the concept of agile robot cell control that requires fast reaction to external events, e.g., based on camera or force sensor feedback. Ensuring safety during robot and human collaboration is also critical. It is essential for the proposed system to react fast to external triggers.

In this experiment, we used the same measurement setup as in Sec. 7.1. We generated robot traffic at 10 Gbps and sampled the latency every 1 ms. In the beginning, we loaded 3.4K trajectory points to the switch and then started the operation. In every 1 second, we add 1.6K new TPs and remove 10K outdated TPs, illustrating the case when the switch is only used as a playback buffer, and the trajectory segments are loaded incrementally, while the old points are removed. Note that inserting a trajectory point with 6 joints requires the insertion of 12 entries into two exact-match tables. According to realistic scenarios, a trajectory normally contains 5-10 points in a second. Fig. 12 illustrates the latency samples and their moving average (on the bottom), and also shows the number of trajectory points (black) loaded into the switch in time, marking the insertion (blue) and removal (red) phases (on the top). One can observe that the insertion does not affect the packet processing latency in this scenario.

8 Discussion on Possible Deployment

Apart from the theoretic aspect and the successful proof study that the proposed system is feasible to deploy, the possibility of a real industrial deployment is much dependent on the cost factors. A simple calculation reveals that a Tofino-based router costs approx. 9500 USD and it can serve up to 500-1000 robots in parallel which means that the cost of controlling a robot is less than 10-20 USD. It is less than applying mini PCs for hobby use, e.g., Raspberry Pis, and far less than certified industrial robot controllers or routers. The energy consumption of the proposed setup is expected to be much lower than the sum of industrial routers and robot controllers. A network device has better transported traffic per watt ratio than a general purpose computation device that the current robot controllers contain. Though in industry, usually low performance, but reliable old CPUs are applied. According to [3] a programmable switch will result in about 14% extra cost, compared to a non-programmable one, due to the larger area requirement for transistors. It is an interesting aspect if energy saving can be achieved by switching non-working elements on and off. One can observe that the capabilities of a Tofino are far more than what is required for the robot control use case. The utilization of the device can be improved by only

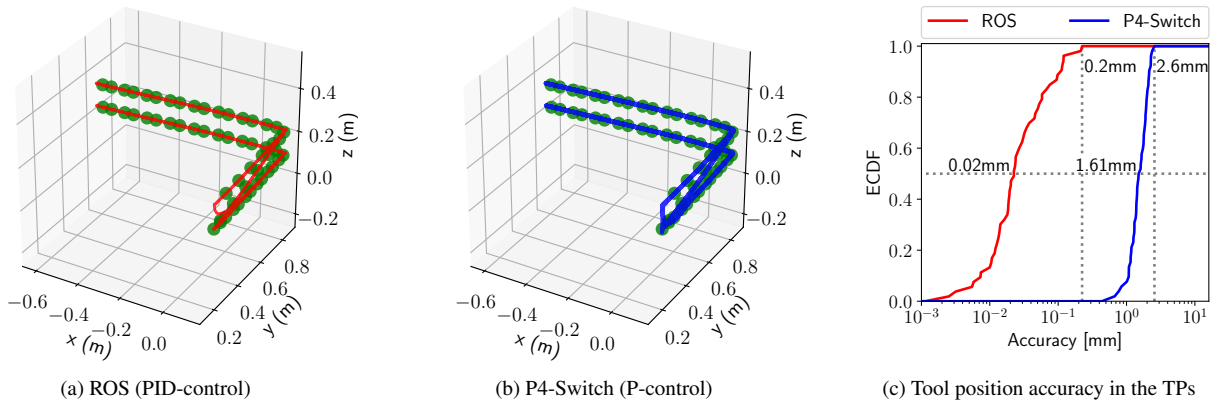(a) ROS (PID-control) (b) P4-Switch (P-control) (c) Tool position accuracy in the TPs

Figure 11: Path of the robot arm tool in the Cartesian-space and the observed accuracy in the TPs.
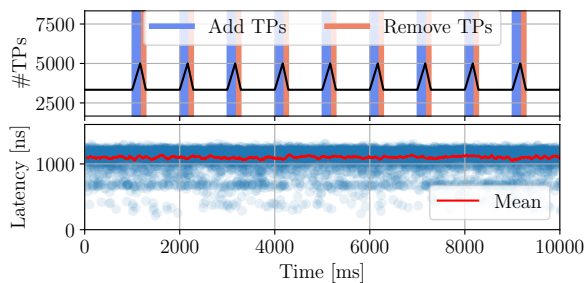

Figure 12: Dynamic insertion and removal of trajectory points.

dedicating a part, e.g., a quarter of the switch to robot control while other parts can work on other tasks (e.g., routing).

The current workflow of a robot is that when it is switched on, it starts streaming out the internal status messages at a constant rate. Production cells are expected to operate 24 hours a day, so little can be done dynamically, apart from the fact that the default power consumption is significantly lower than current systems. The typical power consumption of a Tofino switch is around 110 W [16], while an average server requires 400-600 W. A modern server CPU alone can consume 165 W (Intel Xeon Gold 6348H Processor) at full load. If we compare the costs of purchasing a server with similar processing power and memory to the cheapest P4-Switch, the difference is not too significant. For a brief discussion on x86 alternatives see Appx. B. Also note that this is the first commercially available version of the Tofino switch, and as more and more new models appear and become more available, prices are expected to drop.

Considering the edge-cloud deployment scenario mentioned in Sec. 1, offloading computations that are simple but have real-time requirements that cannot be satisfied in a virtualized environment also have practical benefits. In this case,

distributed service cards or smart NICs with P4 programmability could be more cost effective solutions than a Tofino-based switch. They cost around 1500-3000 USD, also enables line rate (10-40 Gbps) processing with sub-millisecond response time, and have a typical power consumption of 20-50 W.

9   Conclusion

In this paper, we have introduced the first in-network control system that uses P4-programmable network devices for not just triggering events based on threshold values, but to do low-lever real-time velocity control for highly delay-sensitive robotic arms that can be used in industrial automation. With several experiments, we have proved that our system satisfies the most crucial factors of industrial robot control. We measured the latency and observed that it meets the requirements needed for real-time control even during the constant insertion and deletion of lookup table entries. We witnessed a maximum of an 8 ms time shift in the worst-case scenario between synchronous robots, making them fully capable of collaboration. We evaluated the end-position precision per joint to be under $0.5\mu m$ for a 1 m long robot arm, while the accuracy along the whole trajectory to be lower than 2.6 mm in the worst-case.

Acknowledgment

References

[1] Mehdi Bennis, Mérouane Debbah, and H. Vincent Poor. Ultra-Reliable and Low-Latency Wireless Communication: Tail, Risk and Scale. *CoRR*, abs/1801.01270, 2018.

[2] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[3] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. *SIGCOMM Comput. Commun. Rev.*, 43(4):99–110, August 2013.

[4] Fabricio E Rodriguez Cesen, Levente Csikor, Carlos Recalde, Christian Esteve Rothenberg, and Gergely Pongrácz. Towards low latency industrial robot control in programmable data planes. In *2020 6th IEEE Conference on Network Softwarization (NetSoft)*, pages 165–169. IEEE, 2020.

[5] David Coleman, Ioan Alexandru Sucan, Sachin Chitta, and Nikolaus Correll. Reducing the barrier to entry of complex robotic software: a moveit! case study. *ArXiv*, abs/1404.3785, 2014.

[6] Comau 5G deployment. https://www.ericsson.com/en/reports-and-papers/ericsson-technology-review/articles/industrial-automation-enabled-by-robotics-machine-intelligence-and-5g, 2017.

[7] P. Danielis, J. Skodzik, V. Altmann, E. B. Schweissguth, F. Golatowski, D. Timmermann, and J. Schacht. Survey on real-time communication via ethernet in industrial automation environments. In *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, pages 1–8, 2014.

[8] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. Moongen: A scriptable high-speed packet generator. In *Proceedings of the 2015 Internet Measurement Conference*, IMC '15, page 275–287, New York, NY, USA, 2015. Association for Computing Machinery.

[9] Y. Guo, X. Hu, B. Hu, J. Cheng, M. Zhou, and R. Y. K. Kwok. Mobile cyber physical systems: Current challenges and future networking applications. *IEEE Access*, 6:12360–12368, 2018.

[10] ISO: International Organization for Standardization. 1998. Manipulating industrial robots – Performance criteria and related test methods, NF EN ISO9283. https://www.iso.org/standard/22244.html, 1998.

[11] B. Kehoe, S. Patil, P. Abbeel, and K. Goldberg. A survey of research on cloud robotics and automation. *IEEE Transactions on Automation Science and Engineering*, 12(2):398–409, April 2015.

[12] Thomas Kohler, Ruben Mayer, Frank Dürr, Marius Maaß, Sukanya Bhowmik, and Kurt Rothermel. P4cep: Towards in-network complex event processing. In *Proceedings of the 2018 Morning Workshop on In-Network Computing*, pages 33–38, 2018.

[13] D. W. McKee, S. J. Clement, J. Almutairi, and J. Xu. Massive-scale automation in cyber-physical systems: Vision amp;amp; challenges. In *2017 IEEE 13th International Symposium on Autonomous Decentralized System (ISADS)*, pages 5–11, March 2017.

[14] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. Ros: an open-source robot operating system. In *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA) Workshop on Open Source Robotics*, Kobe, Japan, May 2009.

[15] Jan Rüth, René Glebke, Klaus Wehrle, Vedad Causevic, and Sandra Hirche. Towards in-network industrial feedback control. In *Proceedings of the 2018 Morning Workshop on In-Network Computing*, pages 14–19, 2018.

[16] Nik Sultana, John Sonchack, Hans Giesen, Isaac Pedisich, Zhaoyang Han, Nishanth Shyamkumar, Shivani Burad, André DeHon, and Boon Thau Loo. Flightplan: Dataplane disaggregation and placement for p4 programs. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, April 2021.

[17] Universal Robot 5e. https://www.universal-robots.com/products/ur5-robot/, 2020.

[18] URCap. https://www.universal-robots.com/about-universal-robots/news-centre/launch-of-urcaps-the-new-platform-for-ur-accessories-and-peripherals/, 2014.

[19] URScript. https://www.universal-robots.com/how-tos-and-faqs/how-to-ur-how-tos/ethernet-socket-communication-via-urscript-15678/, 2017.

[20] Jonathan Vestin, Andreas Kassler, and Johan Åkerberg. Fastreact: In-network control and caching for industrial control networks using programmable data planes. In

2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA), volume 1, pages 219–226. IEEE, 2018.

(a) Stage occupancy.

(b) SRAM usage.

(c) TPs to be stored.

(d) Min. episode time.

Figure 13: Resource usage of different setups with variable number of robot arms, different trajectory granularity and control plane speed.

A Scalability estimation

We have carried out various micro-benchmark measurements to estimate the scalability of the proposed in-network robot control method in terms of both computational, memory resources and the speed of interaction between data and control planes. Fig. 13a-13b show the first experiment group where the number of trajectory points stored by the switch for each robot is varied. We consider three settings: 50, 100 and 200 TPs/robots. Note that in an average case, a trajectory consists of 10 TPs in every second. These numbers can be interpreted in two ways: 1) this is the number of TPs in the entire trajectory of a given robot, 2) the TPs related to a trajectory episode as discusses in Sec. 7.5 (Note that an epsiode of n TPs requires space for storing at least $3n$ TPs in the pipeline: expired episode to be deleted, active episode that is under execution, upcoming episode that will be executed after the active one). The TPs are stored in exact tables of the pipeline that are mapped to the SRAM. One can observe that both the number of stages and the SRAM usage scale linearly with the number of robot arms. The increase in SRAM usage expresses the rising number of table entries (6 entries in two tables for each TP). Note that the SRAM usage could be the same or similar in other P4-targets (e.g., smartNICs, DSCs). However, the increase in the number of stages is directly related to the physical structure of underlying P4-device, and could vary from target to target. In our case, SRAM is distributed among stages, and if the table is too large, it is spread among multiple stages, increasing the stage occupancy. Note that the P4-switch we used for evaluation is able to store at most 50K TPs without any limitation. The TCAM usage of the proposed method is limited and predictable. Tables used for approximating the calculations in the PID-like controller are mapped to the TCAM area whose size only depends on the required control precision (Sec. 5.3).

Fig. 13c-13d focus on the dynamic use case discussed in Sec. 7.5, showing the relationship between resource usage (#TPs), the number of robots, the length of episodes (t_{ep}), the granularity of trajectories (r_{tp}: normal usage with 10 TPs/s; fine-grained movement with 43 TPs/s) and the time (t_{upd}) needed the control plane for updating tables storing TPs in data plane. Note that t_{upd} in the figure illustrates the time needed for adding and removing 1.6K TPs (2x10K entries), and according to our measurements the update time scales linearly with the number of TPs, but it cannot go below 1 ms. One can observe that in this dynamic scenario the speed of the control plane determines both the minimum length of a trajectory episode and the maximum number of robots to be controlled for a given r_{tp}. In our prototype control plane t_{upd} is almost 300 ms, and thus for r_{tp}=10 TPs/s 500 robot arms can be controlled with $t_{ep} \geq 1s$. One can also see that it requires less memory resources than the 50K limit and thus the speed of the control plane has become the bottleneck in this case, limiting the number of robots to be integrated.
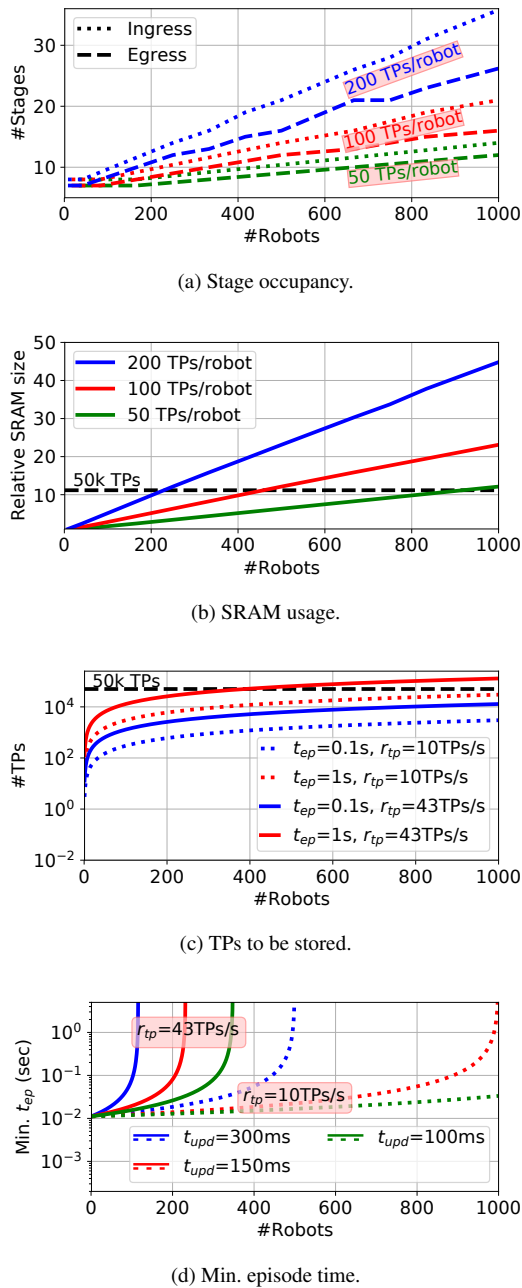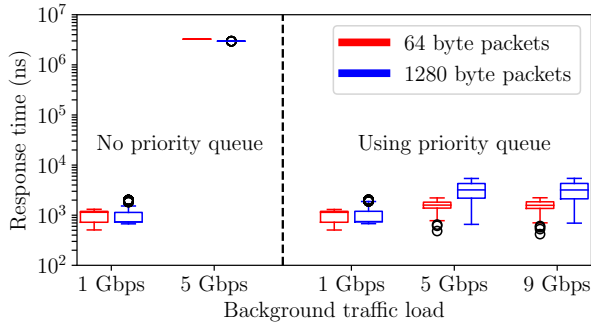
Figure 14: The observed response times in case of a 5Gbps bottleneck

B Comparison to x86

We have shown in the previous section that a single P4-switch can be scaled up to control 500 or even 1000 robot arms, depending on the use cases and settings. However, the low-level velocity vector calculation can also be separated from the industrial controller and offloaded to a dedicated computer in a traditional scenario. In this section, we consider a distributed ROS deployment where the low-level control is coordinated by a robot-driver node in ROS. For each robot arm, a dedicated process is executed to receive status messages, perform the calculations and send velocity commands. Robot-driver nodes require real-time Linux kernel to ensure the timing requirements. We evaluated the driver node of UR5e in a multi-core server equipped with two CPUs (2x Intel Xeon CPU E5-2630 2.30GHz 6C/12T, 32GB RAM). A single control process resulted in 0.19 CPU usage (out of 12, the number of logical cores) in idle state which went up to 0.42 after the robot arm was connected. The CPU usage scaled linearly with the number of robot arms. The CPU limit was reached with 45 emulated robot arms after that ROS processes started interfering each other. The total system load was around 95%.

C Interference with regular network traffic

In Sec. 7.1, we have shown that the response time in underloaded situations is around 1-2μs. Though we think that the separation of control and regular traffic could be possible in most environments, in this section we investigate how regular traffic with different load level affects the processing of control messages. To make the interference more visible, the port rates of the switch are limited to 1Gbps, 5Gbps and 9Gbps. 90% of the test traffic is regular traffic (i.e., non robot control packets) while the remaining 10% consists of robot control messages. The load level is varied from 1Gbps to 9Gbps. The same testbed is used as in Sec. 7.1.

In Fig. 14, we have created an artificial bottleneck of 5Gbps by rate limiting the used egress port. The left side of the figure depicts the case when the regular and robot control traffic is not separated from each other. The packet size in the regular traffic is either 64 or 1280 bytes, marked with red or blue, resp. One can observe that when the arrival rate is 1 Gbps which is much smaller than the bottleneck capacity, the response time is around 1μs as in our previous analysis. Note that no packet loss is experienced in this case. However, when the arrival rate of the test traffic is increased to 5Gbps, the outgoing port starts being congested, packets accumulate in the buffer and thus the observed latency of robot control packets significantly increases (3×10^6 ns = 3ms) due to queueing and a part of the packets is lost. One can note that the increased response times and packet losses degrade the performance of our robot control method, making it unreliable. With an arrival rate of 9Gbps, almost all robot control packets are lost due to congestion. Regular traffic with high intensity has a clear impact on the robot control traffic if they share the same buffer.

However, most P4 programmable devices allow to define multiple queues for each egress port and apply strict priority scheduling between them. As depicted on the right side of Fig. 14, directing regular and robot control traffic into two separate buffers, applying strict priority scheduling between them and giving higher priority to robot control traffic can easily solve the problem of interference. Even in extreme congestion situations (5Gbps or 9Gbps background load) the response times still remain in sub-millisecond order with zero packet loss.

Note that we have obtained similar results for other bottleneck capacities.

Though the first generation P4-programmable hardware targets including smartNICs and switches have well-known limitations, they can still be used for offloading real-time computational tasks. We think that the next generation of such devices that are on the horizon will give a momentum to the use of in-network computing and enable supporting various applications that require ultra-low latency, high-throughput real-time and/or predictable performance. We believe that real-time cloud and edge cloud applications like robot control are one of the potential use cases that can benefit from in-network computation.

Enabling IoT Self-Localization Using Ambient 5G Signals

Suraj Jog[†], Junfeng Guan[†], Sohrab Madani[†], Ruochen Lu[*], Songbin Gong[†], Deepak Vasisht[†], Haitham Hassanieh[†]

University of Illinois at Urbana Champaign[†], University of Texas at Austin[*]

Abstract – This paper presents *ISLA*, a system that enables low power IoT nodes to self-localize using ambient 5G signals without any coordination with the base stations. *ISLA* operates by simply overhearing transmitted 5G packets and leverages the large bandwidth used in 5G to compute high-resolution time of flight of the signals. Capturing large 5G bandwidth consumes a lot of power. To address this, *ISLA* leverages recent advances in MEMS acoustic resonators to design a RF filter that can stretch the effective localization bandwidth to 100 MHz while using 6.25 MHz receivers, improving ranging resolution by 16×. We implement and evaluate *ISLA* in three large outdoors testbeds and show high localization accuracy that is comparable with having the full 100 MHz bandwidth.

1 Introduction

Recent years have witnessed a tremendous growth in the number of connected IoT devices, with surveys projecting up to 31 billion deployed IoT nodes by 2030 [38]. With such ubiquitous deployment of IoT nodes, the ability to localize and track these nodes with high accuracy is essential for many applications. For example, in data driven agriculture, it can enable real time micro-climate monitoring and livestock tracking [39]. In smart cities, IoT sensors are deployed throughout the city for tasks such as air quality monitoring, tracking buses, trains, and cars, and monitoring the structural health of infrastructure [22]. In the era of Industry 4.0, it can also enable wide area inventory tracking and facilitate factory automation [24].

Today, the most prevalent outdoors localization technology is GPS which is mainly used in cars and mobile phones. However, off-the-self GPS chips can consume about the same power as the entire IoT device, thus reducing the battery life to half in addition to the extra hardware costs [5]. Due to this, past work has proposed the use of cellular networks or dedicated IoT base stations for localization [9, 27]. These solutions, however, either achieve very low resolution of 100s of meters [9, 18] or require active participation of the base stations to jointly compute the location or tightly synchronize the base stations [27, 40, 45]. Realizing such solutions in practice requires the cooperation of cellular providers to bear the additional cost of modifying the base stations and a back end server to support the localization feature.

In this paper, *we ask whether an IoT device can accurately localize itself simply by listening to ambient 5G cellular signals, without any coordination with the 5G base stations?* Doing so would allow us to easily deploy self-localizing IoT nodes is wide areas without the need to modify the cellular base stations or deploy new base stations for localization.

5G cellular networks present unique opportunities for enabling accurate localization. First, the small cell architecture in 5G networks will lead to a very high density of 5G base stations, with up to 40 to 50 base stations deployed per square km [15], thereby allowing us to leverage more anchor points in the network for increased localization accuracy. Second, the 5G standard is designed to support very high data rates and can have OFDM signals spanning up to 100 MHz in bandwidth in the sub-6 GHz frequency range, and up to 400 MHz bandwidth in the mmWave frequency range [37]. Such large bandwidth can be used for accurate localization. To see how, consider the 5G OFDM signal shown in Fig. 1(a) where data bits are encoded in *N* frequency subcarriers. We can use the preamble which contains known bits to compute the channel impulse response (CIR) by taking an inverse FFT. The CIR in Fig. 1(a) shows the Time-of-Flight (ToF) of different signal paths. Estimating the ToF from few base stations allows us to localize the device. The larger the bandwidth of the signal, the higher the resolution. In fact, we can achieve a resolution of 3 meters for 100 MHz and 0.75 meters for 400 MHz signals.[1]

Leveraging these opportunities, however, is challenging since power-constrained and low-cost IoT nodes cannot capture the large bandwidth of the 5G signals. They are equipped with low-power and low-speed Analog-to-Digital Converters (ADCs) that can only capture a narrow bandwidth. In fact, while IoT has been one of the cornerstone applications in the design of 5G, it is only supported in narrowband chunks for low data rate applications [2, 3]. Therefore, while the 5G standard does allocate higher bandwidth (up to 400 MHz) for mobile broadband and high data rate applications, IoT nodes can capture only a very small fraction of this bandwidth ($\sim 20\times$ smaller [37]). As a result, they significantly lose out on the ToF resolution that was made possible by the high bandwidth 5G signals as shown in Fig. 1(b). Moreover, it is infeasible to measure the absolute time-of-flight without any coordination or synchronization with the base stations.

In this paper, we present *ISLA*, a system that enables **I**oT **S**elf-**L**ocalization using **A**mbient 5G signals. *ISLA* does not require any coordination with or modifications to the base stations. The key enabler of *ISLA* is the use of MEMS (micro-electro-mechanical-system) acoustic resonators. Past work [11, 12] has demonstrated that we can use such MEMS resonators to design new kinds of RF filters that look like a spike-train in the frequency domain, as shown in Fig. 1(c). To understand how we can leverage such MEMS spike-train filters, consider the 5G OFDM signal shown in Fig. 1(a).

---

[1]The resolution is computed as c/B where c is the speed of light and B is the bandwidth of the signal.
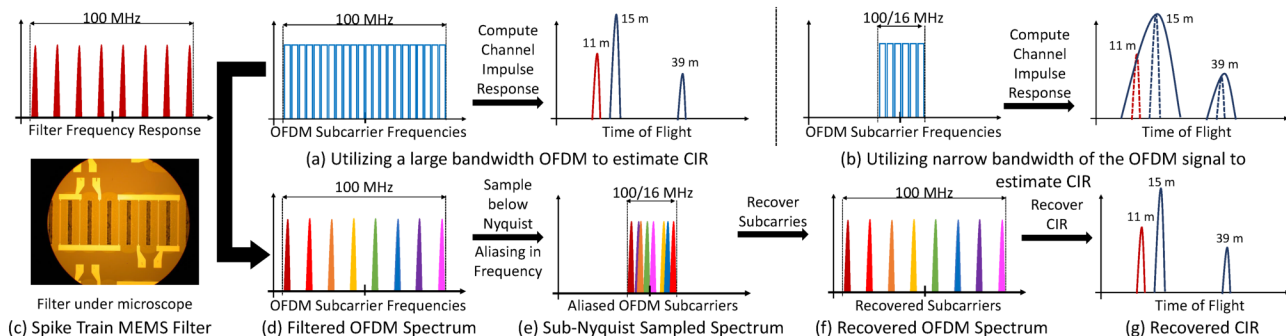
Figure 1: *ISLA*'s pipeline. (a) wideband OFDM signal and its corresponding CIR. (b) narrowband OFDM signal and its corresponding lower resolution CIR. (c) *ISLA*'s spike train MEMS filter that sparsifies the wideband signal. (d-f) follow the signal journey through *ISLA*'s pipeline that recovers the original CIR.

Passing this signal through the filter allows us to keep a few subcarriers of the wideband OFDM symbol while suppressing all other subcarriers as shown in Fig. 1(d). There are two important features of the resulting signal: (1) Since the remaining subcarriers that are passed by the filter span the entire wideband, we should, in principle, be able to recover the channel impulse response at the same high resolution of the original signal. (2) Since the remaining subcarriers create a sparse signal in the frequency domain, it should be possible to recover these subcarriers by sampling the signal below the Nyquist sampling rate using the same low-power low-speed ADCs on the IoT nodes.[2]

However, recovering the channel impulse response from a signal sampled with the low-speed ADCs is non-trivial. First, sampling the signal below the Nyquist rate leads to aliasing in the frequency domain as shown in Fig. 1(e). Some subcarriers might collide by aliasing on top of each other making it hard to recover these subcarriers. Past work in sparse recovery addresses this problem by using two co-prime subsampling rates [16]. Unfortunately, we do not have the flexibility to choose co-prime subsampling factors. In fact, since the number of OFDM subcarriers in the 5G standard is a power of 2 (e.g. 1024, 2048, 4096), we can only subsample the signal by powers of 2 otherwise the values of the subcarriers will be corrupted as we prove in section 5.[3] To address this, we carefully co-design the MEMS hardware with the recovery algorithm. In particular, we jointly optimize the filter shape (spacing between peaks, width of each peak, frequency span) with the subsampling rate to minimize the number of colliding OFDM subcarriers as we describe in detail in section 5.

Second, the recovered OFDM subcarriers are not uniformly distributed across the wideband bandwidth. This is because non-idealities in the MEMS filter make it hard to design a uniform spike train like the one shown in Fig. 1(c). As a result, we can no longer recover the CIR using standard super-resolution algorithms like MUSIC with spatial smoothing [21, 44] as they require uniform measurements. Instead, we formulate an inverse optimization problem that accounts for non-idealities

and optimizes the CIR in the continuous time domain to achieve super resolution as described in Sec. 5.

Finally, while the above can provide very precise ToF measurements, these ToF estimates are not going to capture the true time taken by the signal to travel between the base station and the IoT device. This is because the 5G base stations are not time-synchronized with each other or the IoT device. To localize the device without any synchronization with the base station, *ISLA* leverages a second antenna on the receiver to compute the differential ToF of the propagation paths. While the absolute ToF measurements are corrupted by synchronization offsets, these offsets are constant across the 2 antennas on the IoT node, and hence can be eliminated by subtracting the measurements from the 2 antennas. Using this differential ToF at the IoT receiver, we show in section 7 that with measurements from four or more base stations, the IoT device can localize itself regardless of its orientation. We integrate our approach into a full system that addresses additional system challenges such as figuring the base station ID and accounting for carrier frequency offsets.

Evaluation: We implemented and evaluated *ISLA* indoors for microbenchmarks and outdoors for overall localization performance. We ran experiments in three outdoor settings:(1) Between campus buildings (52 m×85 m), (2) a large parking lot (240 m×400 m), and (3) an agricultural farm (480 m×860 m). We use USRP X310 radios as base stations that can transmit high-bandwidth packets of 100 MHz. Our custom IoT nodes are equipped with 2 antennas and subsample the 5G signals at 6.25 MS/s which is 16× below the Nyquist rate. We fabricated a MEMS spike-train filter operating at a center frequency of 400 MHz and used it to demonstrate accurate reconstruction of the channel impulse response. However, due to significant interference at the 400 MHz band outdoors in our city, we ran experiments at 1 GHz and applied the filter response in digital. Our results reveal that with 5 base stations in range, *ISLA* can achieve a median accuracy of 1.58 m on campus, 17.6 m in the parking lot, and 37.8 m in the farm where the IoT node can be as much as 500 meters away from most base stations. For the parking lot testbed, the accuracy improves to 9.27 m with 15 base stations and 4.26 m with 25 base stations in range. We compare *ISLA*'s localization

---

[2]Note that the MEMS filter is passive and does not consume any power.
[3]For example, for a 100 MHz OFDM signal, we can only sample at 50 MS/s (2×), 25 MS/s (4×), 12.5 MS/s (8×), 6.25 MS/s (16×), ...

approach with several baselines [9, 21, 43] and show up to $4-11\times$ higher localization accuracy. Finally, we show that *ISLA* achieves a comparable performance to having a full 100 MHz receiver while using a $16\times$ lower sampling rate.

Contributions: We make the following contributions:

- We present, to the best of our knowledge, the first system that allows IoT nodes to localize themselves using ambient 5G signals without any coordination with the base stations.
- We demonstrate the ability to reduce the sampling rate by $16\times$ while retaining the benefits of high bandwidth 5G signals by leveraging recent advances in MEMS RF filters.
- We implement and evaluate *ISLA* to demonstrate accurate localization in 3 outdoor settings.

2 Related Work

Localization has been extensively studied in cellular, WiFi, and IoT networks. Our work differs from past research in that it is the first to enable self-localization using ambient 5G signals without requiring coordination with the base stations.

A. Cellular Based Localization: Several studies [9, 17, 18, 29, 33] have proposed to use nearby cell tower information and statistics in order to localize a mobile device. These methods, however, have a median accuracy of around 100 to 500 meters, and are mostly useful for very coarse localization. To improve localization accuracy, [4, 35] propose to combine WiFi APs with cellular base stations. Despite their relatively higher accuracy, these methods require fingerprinting the surroundings and as such require extensive training and do not generalize to new locations. More recent work exploits massive MIMO and millimeter wave for localization in 5G [30, 31, 42]. However, all of this work requires coordination with base stations and assumes the devices can capture the entire bandwidth of the 5G signals which does not work for IoT devices.

B. IoT Based Localization: [5] leverages TV whitespaces to achieve high localization accuracy for LoRA IoT devices. However, it requires all base stations to be tightly synchronized at the physical layer (time and phase) in order to measure TDoA (Time Difference of Arrival). Recent work [27] designs low power backscatter devices that leverage LoRa for localization to achieve high accuracy. However, the system mainly targets indoor applications where software radios can be deployed as base stations to sample the I/Q of the signal and localize the IoT node. Moreover, its current system design [27] supports only a single node. The authors of [34] propose an outdoors localization technique for SigFox IoT devices based on fingerprinting. However, as mentioned earlier, fingerprinting requires constant training and cannot scale to new environments. Finally, there is a lot of work on using UWB or RFID nodes for localization [10, 13, 41]. However, these works focus on indoors and short range as the range of UWB and RFIDs is limited to 10-30 meters [7, 14].

C. IoT Self-Localization: LivingIoT [19] enables self-localization on IoT nodes. It designs a miniaturized device that can be carried by a bumblebee and uses backscatter for communication. The node localizes itself by extracting the angle to the Access Point from the amplitude measurements using an envelop detector. The technique, however, requires the APs to switch the phase across two antennas to change the received amplitude at the IoT node, and hence, cannot be applied to 5G without modifying the base stations. [26] enables self-localization by placing a camera on a WISP RFID but only operates within a range of 3.6 m from the RFID reader.

D. WiFi Based Localization: There has been a lot of work on indoor localization using WiFi [6, 21, 25, 32, 40, 43, 44, 46, 47]. The closest to our work are [21, 40, 43] which estimate the channel impulse response (CIR) and time of flight (ToF) from the WiFi access point (AP). Chronos [40] hops between WiFi channels to compute the CIR at high resolution. However, it requires tight timing coordination with the AP to compensate for carrier frequency offset (CFO) and ensure phase coherence across the measurements. *ISLA*, on the other hand, captures measurements from many frequencies across a wideband without hopping by using the MEMS filter, and hence, does not require any coordination with the base stations. SpotFi [21] combines measurements across antennas with large WiFi bandwidth to separate Line of Sight (LoS) path from multipath reflections in the CIR using MUSIC along two dimensions: ToF and Angle of Arrival (AoA). mD-Track [43] also incorporates Doppler shifts and Angle of Departure (AoD) in addition to ToF and AoA and iteratively refines the CIR to achieve a better estimate of the LoS path. In section 10, we adapt SpotFi's and mD-Track's CIR estimation algorithms to our setting and demonstrate that *ISLA*'s algorithm achieves $4-11\times$ higher accuracy. It is worth noting, however, that for our application, these past works cannot benefit from the doppler or AoA/AoD dimensions.

E. MEMS Filter: Recent work has used MEMS spike-train filters for the application of wideband spectrum sensing [12]. However, [12] can only detect signal power at different frequencies and cannot recover complex I and Q samples needed for estimating the CIR. Furthermore, [12] deals with collisions resulting from aliasing by using co-prime sub-sampling rates. Such approach does not apply in the context of 5G OFDM signals, since, as we show in section 5 the sub-sampling factor can only be a power of 2. *ISLA* instead co-designs the hardware filter together with sampling rate to avoid collisions.

3 Background

A. Spike-Train MEMS Filters: Our work builds on recent advances in MEMS RF filters. MEMS filters can work between a few MHz and 30 GHz and can be integrated with ICs to form a chip-scale RF front-end solution for IoT devices. Past work on MEMS RF filters optimize for filters with a single passband [36, 48], however, the MEMS filter used by

ISLA leverages MEMS resonators that have an assortment of equally spaced resonance frequencies to create a spike train in the frequency domain as shown in Fig. 1(c).

A MEMS filter works by leveraging the inverse piezoelectric effect to convert RF signals into acoustic vibrations for filtering and processing. It then converts acoustic waves in the device back to the RF signals through piezoelectric effect. In this process, the frequency filtering is achieved because not all frequencies can be efficiently converted between RF and acoustic domains. Frequencies that match the resonance frequencies of the piezoelectric structure can go through the conversions with little loss, while other frequencies are filtered out. Hence, the spike train frequencies can be designed by changing the dimension of the piezoelectric material in the MEMS device as well as the placement of electrodes shown under the microscope in Fig. 1(c).

B. Wireless Channel Impulse Response (CIR): The wireless channel can be modeled as the superposition of the signal along all the different paths it takes to travel from the transmitter to the receiver. The channel at frequency f_i can be written as: $h_i = \sum_{l=1}^{L} a_l \exp^{-j2\pi f_i d_l/c}$, where L is the number of propagation paths between the transceivers, d_l is the distance traversed by path l, a_l is the complex path attenuation of path l, and c is the speed of light.

In OFDM systems, data is transmitted over multiple frequency subcarriers $\{f_0, \ldots f_{N-1}\}$. If the frequency spacing between these subcarriers is Δf, then the bandwidth spanned by the signal is $B = \Delta f \times (N-1)$. Now, given the channel measurements $\{h_0, \ldots h_{N-1}\}$ across these frequencies, the Channel Impulse Response (CIR) can be computed as the inverse FFT of the channel measurements.

$$CIR(\tau) = \sum_{n=0}^{N-1} \left(\sum_{l=1}^{L} a_l \exp^{-j2\pi \frac{d_l}{c} f_n} \right) \exp^{j2\pi \tau f_n} \qquad (1)$$

where $\tau = \{\frac{0}{B}, \ldots \frac{(N-1)}{B}\}$ seconds. There are two important things to note here. First, the resolution in Time-of-Flight in the CIR is $1/B$ seconds, that is inversely proportional to the bandwidth B. Hence, larger bandwidth results in higher ToF resolution and more accurate ranging. Second, the maximum unambiguous ToF that can be measured from the CIR is $\frac{(N-1)}{B} = 1/\Delta f$ seconds. This means, if some physical propagation path in the environment has ToF $> 1/\Delta f$ then it would alias and appear at a different tap value in the estimated CIR in Eq. 1. For 5G OFDM signal with $B = 100$ MHz bandwidth and $\Delta f = 60$ kHz , we have a resolution of 10 ns (3 meters) and a range of 16.6 μs (5 km).

4   System Overview

ISLA enables self-localization on narrowband IoT devices by leveraging the MEMS spike-train filter to capture ambient wideband 5G signals. *ISLA* consists of 3 main components:

(1) Capturing the wideband 5G OFDM signal using the MEMS filter: The received 5G signal is passed through the
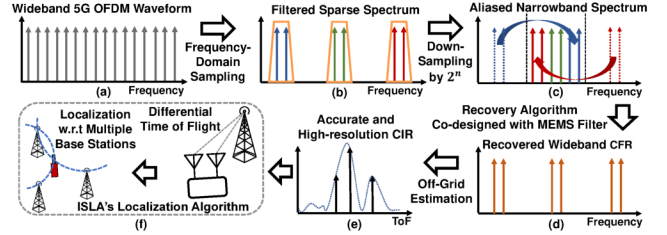

Figure 2: Overview showing the flow of *ISLA*'s system

MEMS filter which samples the OFDM symbol in the frequency domain. Specifically, the MEMS filter passes the OFDM frequency bins that align with the filter passbands while suppressing all other frequency bins. The resulting output from the filter is a sparse spectrum as shown in Fig. 2(b). This sparse signal is then subsampled by the narrowband IoT device significantly below the Nyquist rate (16× lower) which results in aliasing the remaining subcarriers into the narrowband as shown in Fig. 2(c). We co-design the filter hardware with the recovery algorithm to easily reconstruct the wideband OFDM subcarriers as we describe in section 5.

(2) Super-Resolution CIR Estimation: Using the recovered wideband channel measurements, *ISLA* then reconstructs a high resolution Channel Impulse Response (CIR) by leveraging its super-resolution algorithm which estimates the off-grid positions of the propagation paths as described in Section 6. This high-resolution CIR allows *ISLA* to filter out the LoS path from the multipath in the channel for high resolution time-of-flight estimation as shown in Fig. 2(e).

(3) Localization Algorithm: Since the IoT node is not synchronized with the base station, the measured ToF will be corrupted by a timing offset. To address this, *ISLA* leverages two antennas on the IoT device and computes the differential CIR across the antennas to eliminate the synchronization offsets. This results in the locus of the IoT device to lie on a circle that is defined by the locations of the base stations and the angle subtended by the base stations at the IoT device's location, as we explain in Section 7. Thus, by looking at the intersection of such circles, we can accurately infer the position of the IoT device as shown in Fig. 2(f). Finally, we show how to integrate *ISLA* with the 5G-NR standard by addressing additional system challenges in section 8.

5   Capturing 5G Signals Using MEMS Filter

ISLA leverages the MEMS spike-train filters to capture the wideband channel measurements on a narrowband receiver. We explain this sensing process through Fig. 2. Consider a preamble OFDM symbol transmitted from the base station with N subcarrier frequencies at $\{f_0, \ldots, f_{N-1}\}$, shown in Fig. 2(a). Let the received time domain symbol be $x(t)$ and its frequency domain representation be $X(f)$. We have $X(f) = \sum_{n=0}^{N-1} c_n h_n \delta(f - f_n)$, where c_n are the data bits modulated onto the subcarriers and h_n are the channel values at f_n. We want to extract this channel information to compute

the Channel Impulse Response $CIR(\tau)$. Since the preamble bits c_n are known, we can compensate for c_n and compute the $CIR(\tau)$ by taking an IFFT of the channel values h_n. However, this requires capturing the entire bandwidth of the 5G OFDM signal. Our goal is to recover the CIR using a narrowbandwith. To do so, we leverage the MEMS spike-train filter.

The spike-train filter response is made up of uniformly spaced passbands as shown in Fig. 2(b). The spike-train filter serves to sparsify the OFDM symbol by selectively passing subcarriers that fall inside the MEMS passbands, while suppressing all other frequencies. Let the set of frequencies passed by the spike-train be indexed by M. Then, the frequency domain of the signal $\tilde{X}(f)$ ($\tilde{x}(t)$ in the time domain) after passing through the spike-train filter will be $\tilde{X}(f) = \sum_{i \in M} c_i h_i \delta(f - f_i)$.

This sparse spectrum is shown in Fig. 2(b). Next, the IoT receiver subsamples the signal $\tilde{x}(t)$ using a low-speed ADC that samples at a rate $R = B/P$, where B is the bandwidth of the transmitted symbol and P is an integer corresponding to the subsampling factor. Let $y(t)$ be the subsampled signal, that is, $y(t) = \tilde{x}(P \times t)$, and let $Y(f)$ be its frequency domain representation. Then $Y(f)$ is an aliased version of $\tilde{X}(f)$:

$$Y(f) = \sum_{i=0}^{P-1} \tilde{X}(f + iR) \qquad (2)$$

$Y(f)$ will cover a narrow bandwidth equal to R MHz as depicted in Fig. 2(c). The process of aliasing is as follows. Any frequency f_j, $j \in M$, that falls outside the narrowband of the IoT device, will alias onto the frequency bin $\tilde{f}_j$ inside the narrowband after subsampling, such that $f_j - \tilde{f}_j = z \times R$, where z is some integer. Note that for every f_j, we have a unique $\tilde{f}_j$. So given the measurement at the aliased frequency $\tilde{f}_j$, we can potentially recover the channel value h_j at the corresponding unaliased frequency f_j.

However, recovering these channel values from the aliased spectrum is non-trivial because multiple of the frequency subcarriers passed by the spike-train filter may collide by aliasing on top of each other and summing up. This is unfavorable since now we are unable to extract the channel values for any of the colliding frequencies. Past work addresses this by leveraging multiple co-prime subsampling factors, which ensures that the same frequencies don't collide repeatedly.

Unfortunately, we do not have such flexibility to choose any sub-sampling factor here. This is because in order to recover the channel value h_j from the aliased frequency $\tilde{f}_j$, we need to ensure that the complex scaling factor $c_j \times h_j$ encoded on subcarrier f_j remains preserved upon aliasing. This is crucial because the wireless channel information is contained inside this scaling factor. The following lemma states the condition that ensures this:

Lemma 5.1. *For a sub-sampling factor P and N OFDM subcarriers, the complex valued scaling factors for each subcarrier will be preserved upon aliasing if $N = z \times P$, for some integer z, given the aliasing results in no collisions.*

The proof for the above lemma is in Appendix A. Thus, to be able to recover channel values, we are restricted to subsample the signal by an integer factor of N. Further, since the OFDM subcarriers in the 5G standard are set to powers of 2, we can only subsample the wideband signal by powers of 2.

Due to this lack of choice in subsampling factors, we instead shift our focus on designing the spike-train filter such that the frequencies passed by the filter do not collide upon aliasing. We achieve this by leveraging the structured periodic sparsity of the spike-train, and design a filter that ensures no collisions for the given subsampling factor P.

Doing so significantly simplifies our recovery algorithm. In particular, given that (1) the frequency response of the spike-train filter and its collision-free aliasing patterns are known, and that (2) the scaling factors at the frequency subcarriers remain preserved upon aliasing, we can now simply rearrange the frequencies in Y(f) to their corresponding unaliased frequency positions as shown in Fig. 2(d). Further, we can extract the channel values at these unaliased frequencies by dividing the complex scaling factor $c_j \times h_j$ by the known preamble bit c_j. Thus, by leveraging the spike-train filter, *ISLA* is able to extract wideband channel values on a narrow band IoT device. Next, we discuss the design parameters of the spike-train filter that ensures no collisions.

Spike-Train Filter Design: We explain the spike-train filter design with a specific example, shown in Fig. 3(a). Let the wideband transmitted OFDM signal (B MHz bandwidth) be comprised of 32 frequency subcarriers, indexed from -16 to 15, with 0 denoting the carrier frequency bin. From Lemma 5.1, we want the subsampling factor P to divide $N = 32$. So we choose $P = 4$, that is, the IoT receiver subsamples the signal by $4\times$. This implies that the IoT receiver is only able to capture $\frac{N}{P} = 8$ frequency bins centered around the carrier frequency as shown by the shaded region in Fig. 3(a). Let this narrow band set of frequencies be denoted as f_{NB}.

Recall that when you subsample a *B* MHz signal by $P\times$, then all frequency subcarriers spaced by $R = \frac{B}{P}$ MHz will alias onto the same frequency bin in the narrow band spectrum. Here, this translates into all frequencies spaced by 8 subcarriers aliasing onto the same narrowband bin. This is depicted in Fig. 3(a) through the color coding scheme. For instance, the subcarriers at $\{-9, -1, 7, 15\}$ (represented as purple colored) would all appear at frequency bin -1 in the narrow band spectrum upon aliasing. For a given subcarrier k in the narrow band spectrum, that is, $k \in \{-4, \ldots, 3\}$, let us denote the set of subcarriers that would alias into k as I_k. So we have $I_{-1} = \{-9, -1, 7, 15\}$.

The spike-train filter will selectively pass frequency subcarriers in the wideband OFDM signal, which after aliasing can be recovered from the narrow band signal at the receiver. Let the set of frequency subcarriers passed by the spike-train filter be denoted by f_M, where $M \in [-15, \ldots, 16]$. We want the following conditions to hold:

(a) Spike-Train Filter Parameters (b) Spike-Train Filter Frequency Response (c) Aliased Frequency Response
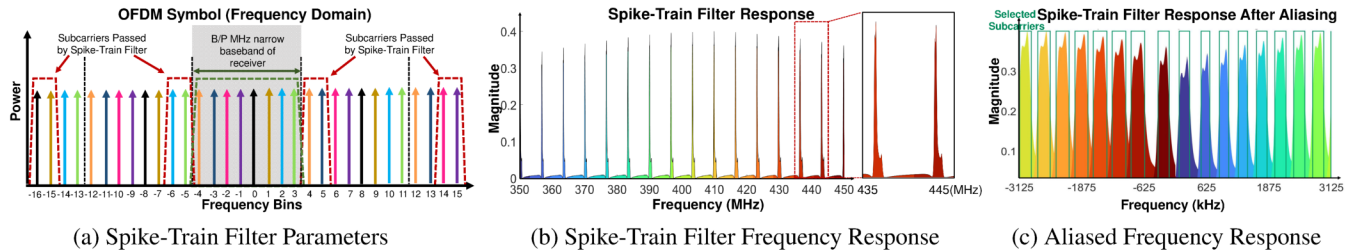
Figure 3: (a) MEMS Filter Parameters that ensure zero collisions while recovering maximum channel information. (b) Frequency response of MEMS spike-train filter. (c) Aliasing pattern of spike-train filter frequency response.

1. *No Collisions:* To ensure that we can successfully recover the wideband channels, no two subcarriers in f_M should alias and collide in the same narrowband frequency bin upon subsampling. To achieve this, the spike-train filter must satisfy: *For any set I_k where $k \in \{-4, \ldots, 3\}$, f_M must contain at most one subcarrier from I_k.*

2. *Extract Maximum Possible Channel Values:* Given that the narrowband spectrum spans 8 frequency subcarriers, this means that the receiver can successfully recover at most 8 channel values after subsampling. In the presence of noise, we want to recover as many channel measurements as possible for robustness. Hence, every narrowband subcarrier in f_{NB} should yield one channel measurement from the wideband signal. This translates to: *For any set I_k where $k \in \{-4, \ldots, 3\}$, f_M must contain at least one frequency subcarrier from I_k.*

   1 and 2 put together, dictates that the spike-train filter should pass *exactly one* frequency subcarrier from each I_k.

3. *Span the Wideband OFDM symbol:* To retain the high ToF resolution, we want the set of frequencies in f_M to span the entire wideband signal.

The above conditions can be met leveraging the structured sparsity in the spike-train filter response. Specifically, we can design three key parameters of the spike-train filter: (1) spacing between consecutive spikes ΔF, (2) width of the spikes ΔS, and (3) the starting frequency subcarrier f_M^0 in the spike-train, to follow Lemma 5.2. We prove in Appendix A that such a filter response satisfies the above conditions.

Lemma 5.2. *Consider an OFDM symbol with N frequency subcarriers, indexed as $\{f_{-\frac{N}{2}}, \ldots, 0, \ldots, f_{\frac{N}{2}-1}\}$ with inter-frequency spacing of Δf, and a narrowband receiver that subsamples by $P\times$. If P^2 divides N, then the ideal filter parameters that meet all three requirements are: (1) $f_M^0 = f_{-\frac{N}{2}}$, (2) $\left(\frac{N}{P^2} - 1\right) \times \Delta f < \Delta S < \frac{N}{P^2} \times \Delta f$, and (3) $\Delta F = \frac{N}{P}(1 + \frac{1}{P}) \times \Delta f$.*

Furthermore, we can achieve the required filter response by designing the topology of the MEMS resonators, which we explain in more details in Appendix B.

In Fig. 3(a), we show the ideal frequency response of the spike-train filter designed with the above parameters as the red dotted line. In theory, such a filter should allow us to leverage all f_{NB} subcarriers to recover the wideband channel measurements from the aliased signal. However, in practice,

MEMS spike-train filters are non-ideal i.e., the roll-off of the passband boundaries are not as sharp as perfect rectangular functions, the spikes are not perfectly equally spaced, and the passband widths are not identical. These imperfections can be observed in the frequency response shown in Fig. 3(b). As a result of these non-idealities, there will still be collisions at the boundary regions of the spikes after aliasing, as shown in Fig. 3(c). To avoid collisions from polluting our CIR estimates, we only consider the subcarriers that do not collide as shown in Fig. 3(c). However, this results in non-uniform sampling of the OFDM subcarriers across the wideband channel. In sec. 6, we show how to leverage *ISLA*'s super-resolution algorithm to recover high resolution CIR estimates from these non-uniform channel measurements.

Tradeoff Between Range and Resolution: Recall from section 3 that the resolution in ToF depends on bandwidth, whereas the maximum unambiguous ToF (range) depends on the inter-frequency spacing between channel measurements. In the 5G OFDM signal with bandwidth $B = 100$ MHz and subcarrier spacing $\Delta f = 60kHz$, *ISLA* is able to retain the high ToF resolution of 10 ns (3 m) by collecting wideband channel measurements that span the entire 100 MHz. However, in doing so, the frequency spacing between the channel measurements in *ISLA* increases, thus reducing the maximum ToF range. Specifically, the frequency spacing increases by $P = 16\times$ in *ISLA*, thus reducing the maximum range from 5 km to 312 meters. This is an issue since now it becomes difficult to identify the LoS path from the CIR for localization. You could have the case where the LoS path is at 200 meters but a reflected path at 400 meters aliases and appears at the bin corresponding to 88 meters in the CIR. Thus, you cannot simply pick the first peak as LoS.

To address this, *ISLA* combines the wideband channel measurements from the spike-train filter, h_M, with the narrowband channel measurements h_{NB} collected at the subcarriers f_{NB}, and formulates a joint optimization with both these channels to estimate the CIR. Since the narrowband channel measurements h_{NB} retain the same subcarrier spacing of $\Delta f = 60kHz$, it increases the effective maximum ToF range back to 5 km, thus resolving the LoS ambiguity in the CIR.
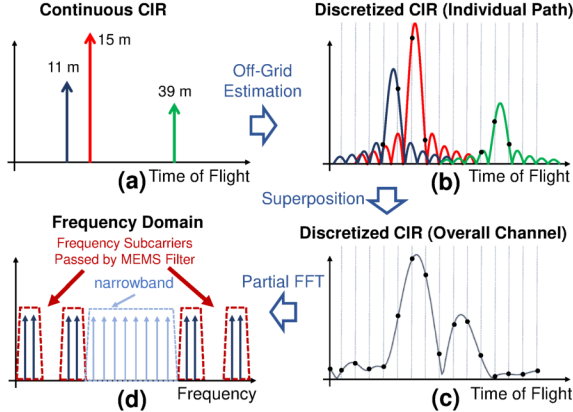
Figure 4: Signal paths to measured channel forward function

6   Super-Resolution CIR Estimation

Here we describe our super-resolution algorithm that can retrieve high resolution ToF estimates τ_l's along with the associated complex attenuations a_l for the L multipath components in the channel. As discussed in Sec. 5, the IoT device can recover channel measurements $h_{tot} = h_M \cup h_{NB}$ at the subcarriers $f_{tot} = f_M \cup f_{NB}$ where f_M are recovered from the spike-train filter and f_{NB} without the filter. Since these channel values are sampled at non-uniformly spaced frequencies, we cannot apply standard super-resolution algorithms like MUSIC with spatial smoothing [21, 44] as they require uniform measurements. Instead, we optimize for the channel impulse response in the continuous time domain by leveraging an off-grid estimation technique that can estimate high resolution ToF values from the channel information.

We begin by framing this as an inverse problem. We start by modeling the forward operator $\mathcal{F}$: $h_{tot} = \mathcal{F}(\tau_1, \ldots, \tau_L, a_1, \ldots, a_L)$, which maps physical path parameters to the wireless channel. $\mathcal{F}$ comprises of the following distinct transformations, as illustrated in Fig. 4:

(1) CIR in Continuous Domain: (Fig. 4(a)) Given path parameters $\{\tau_1, \ldots, \tau_L, a_1, \ldots, a_L\}$, the continuous domain CIR can be written as: $CIR_{cont} = \sum_{l=1}^{L} a_l \delta(\tau - \tau_l)$, with each path represented as an impulse positioned at its respective ToF τ_l, and scaled by its complex attenuation a_l.

(2) Off-Grid Estimation: (Fig. 4(b)) The OFDM symbol spans a bandwidth B MHz and comprises of N subcarriers. Due to this discretization and truncation in the frequency domain, the observed CIR at the receiver will also be discretized, and computed on the grid defined by τ_g, where $\tau_g = \{\frac{0}{B}, \ldots, \frac{(N-1)}{B}\}$. However, as with most natural signals, the ToFs of the physical propagation paths τ_l will rarely align with this discretized τ_g grid, that is, the τ_l's will lie at an off-grid position. As a result, the leakage from the continuous off-grid CIR component from path l to the discrete CIR grid positions at τ_g can be computed as $CIR^l(\tau_g) = a_l \psi_N(\tau_g - \tau_l)$,

where ψ_N is the discretized sinc function defined as:

$$\psi_N(\tau) = \frac{sin(\pi\tau)}{sin\left(\frac{\pi\tau}{N}\right)} \exp\left(-\pi j \left(\frac{N-1}{N}\right)\tau\right) \quad (3)$$

(3) Superposition: (Fig. 4(c)) With multiple propagation paths in the channel, the net observed CIR at the receiver is the sum of the CIR profiles contributed by each propagation path: $CIR^{net}(\tau_g) = \sum_{l=1}^{L} a_l \psi_N(\tau_g - \tau_l)$.

(4) Discrete Fourier Transform: (Fig. 4(d)) Finally, the channel h_{tot} can be computed by sampling the corresponding frequencies f_{tot} from the DFT of the superposed CIR. Let us denote the $N \times N$ Fourier matrix as $\mathbf{F}_N$, and let $\mathbf{V}$ be the matrix that chooses the rows corresponding to f_{tot} from $\mathbf{F}_N$. Then we have: $h_{tot} = \mathbf{V} \mathbf{F}_N CIR^{net}$ where CIR^{net} is a $N \times 1$ dimension vector.

Putting the above four transformations together, the forward operator $\mathcal{F}$ can be expressed as:

$$h_{tot} = \mathcal{F}(\{\tau_l, a_l\}_{l=1}^{L}) = \mathbf{V} \mathbf{F}_N \Psi \vec{a} \quad (4)$$

where Ψ is a $N \times L$ matrix with $\Psi_{i,j} = \psi_N(\tau_i - \tau_j)$, and $\vec{a}$ is a $L \times 1$ vector comprising the complex attenuations a_l for each path. Now that we have the forward operator, the inverse problem to retrieve the path parameters from observed channel vector h'_{tot} can be formulated as a L-2 minimization:

$$\{\tau_l^*, a_l^*\}_{l=1}^{L} = \underset{\tau_1, \ldots, \tau_L, a_1, \ldots, a_L}{\arg\min} \|h'_{tot} - \mathbf{V}\mathbf{F}_N\Psi\vec{a}\|^2 \quad (5)$$

Solving the Optimization: Note that if we are given Ψ, then Eq. 5 becomes a linear optimization problem in $\vec{a}$. Thus, given Ψ, the closed form solution for $\vec{a}$ that minimizes Eq. 5 is $\vec{a} = (\mathbf{V}\mathbf{F}_N\Psi)^\dagger h'_{tot}$, where $\dagger$ represents the pseudo-inverse. Thus the objective function in Eq. 5 can be rewritten as:

$$\{\tau_l^*\}_{l=1}^{L} = \underset{\tau_1, \ldots, \tau_L}{\arg\min} \|h'_{tot} - \mathbf{V}\mathbf{F}_N\Psi(\mathbf{V}\mathbf{F}_N\Psi)^\dagger h'_{tot}\|^2$$

$$\text{s.t.} \quad \tau_l \geq 0 \quad \forall l \in \{1, 2, \ldots, L\} \quad (6)$$

The objective function is now reduced to just the ToF variables τ_l's. This optimization problem is non-convex and constrained, and we use the well-known interior-point method to solve this [8]. For the initialization point to the optimization algorithm, we use approximate ToF values from the CIR computed by taking the inverse FFT of the observed channel h'_{tot}. While these ToF estimates are distorted by the discretization and superpositioning artifacts described previously, it gives a good starting point for the optimization.

Also, note that the number of paths N in the wireless channel is not known a priori. As we keep increasing the number of paths N that the algorithm is initialized with, it keeps finding a better and better fit to the channel data, and after a point, starts overfitting to the noise. In order to avoid overfitting and yet yield accurate estimates for the path parameters, we run the optimization problem multiple times, each time increasing the number of paths it is initialized with by 1. We terminate the algorithm when the decrease in the value of the objective function falls below some threshold ε, and set the current value of N to be the number of paths in the channel.
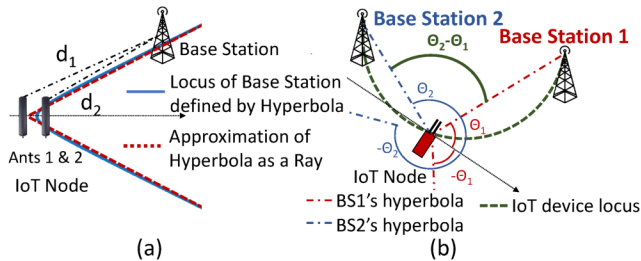
Figure 5: *ISLA*'s Localization Algorithm

7 *ISLA*'s Localization Algorithm

The above off-grid estimation algorithm gives us highly precise ToF estimates for the propagation paths. However, since the 5G base stations are not time synchronized with the IoT device, there is going to be an offset between the sampling clocks in their RF chains. As a result, the measured ToF at the IoT node also includes delays from the sampling time offset (STO) between the different base stations and the IoT node, and hence cannot provide accurate distance estimates.

To address this, *ISLA* leverages two antennas on the IoT node to compute the differential ToF rather than the absolute. The key idea here is that while the absolute ToF measurements are corrupted by synchronization offsets, these offsets are constant across the two antennas on the IoT node. Hence, the offsets can be eliminated by differencing the two measurements. Let the ToF values to the two antennas be τ_1 and τ_2, and their corresponding distances be d_1 and d_2, as denoted in Fig. 5(a). Then the locus of the base station from the IoT device's frame of reference is a hyperbola with the two antennas being the foci, and the difference in distances to the two foci equaling $d_2 - d_1$. At large distances, this hyperbola can be approximated as two rays along the asymptotes of the hyperbola, depicted by the red dashed lines in Fig. 5(a).

By overhearing packets from different base stations, the IoT device can infer the locus of each base station to lie on approximated rays originating from the IoT device's location. This is shown in Fig. 5(b), where base station 1 can lie on the rays at angles θ_1 or $-\theta_1$, and similarly the base station 2 can lie on the rays at angles θ_2 or $-\theta_2$. Both θ and $-\theta$ are possible since there is the ambiguity that the signal might have arrived from the front or the back of the device. Given this, we can see that the angle subtended by the two base stations at the location of the IoT device will be $\|\theta_2 - \theta_1\|$, and this is going to be constant irrespective of the orientation of the IoT node. (There is ambiguity in that the angle subtended can also be $\|\theta_2 + \theta_1\|$, and we will address this shortly).

Given the angle subtended by the base stations and the known locations of the base stations, according to the Inscribed Angle Theorem, we can determine the locus of the IoT device to lie on the arc of a circle, where the line segment connecting the two base stations is the chord and the corresponding inscribed angle is equal to the angle subtended by the base stations. This is illustrated in Fig. 5(b) as the green dashed arc. Leveraging different pairs of base stations, *ISLA*

can draw multiple such arcs and the intersection points of these arcs will give us the IoT device's location.

Sources of Ambiguity: There are some sources of ambiguity that need to be resolved. First, the angle subtended by the two base stations in Fig. 5(b) could also be $\|\theta_2 + \theta_1\|$, and second, the arc drawn with the base stations at the end points could also be pointing towards the north rather than south, as depicted in Fig. 5(b). These ambiguities can be resolved easily by leveraging 4 base stations as anchor points. Keeping one base station common, we have three base station pairs which yields three unique arcs. Only the right configurations of angles subtended and arcs drawn will give us a common intersection point for all three arcs. *ISLA*'s localization algorithm tries all configurations and picks the one where all arcs coincide at the same point.

8 Integrating *ISLA* with 5G-NR Standard

Similar to the LTE standard, the 5G-NR packet consists of 10 subframes, each of duration 1 ms [28]. To allow for coherent packet demodulation, the 5G frame appends known preamble bits on each subframe which enables channel estimation and correction across the entire bandwidth of the 5G channel. Additionally, in the first subframe of the packet, the base station also includes all information required by devices to associate with the network, which comprises of the synchronization signals (PSS and SSS frames) for CFO correction and frame timing, and the Base Station ID. To allow every device in the network to receive this critical information, it is always encoded in the narrowest supported bandwidth of the wideband packet, which is 4.32 MHz in the 5G standard [28].

ISLA's hardware circuit, discussed in Section 9, is designed such that it can switch between capturing the 6.25 MHz narrowband spectrum, or the wideband spectrum via the spike-train filter. *ISLA* begins by capturing the first subframe of the 5G packet through its narrowband RF path, and extracts the synchronization frames and base station ID encoded in the narrowband subcarriers of the wideband packet. Using publicly available databases like [1], *ISLA* can retrieve the location of the Base Station given its ID. The synchronization frames help eliminate coarse CFO and SFO. From the subsequent subframes, *ISLA* first estimates the narrowband channel, and then switches to the RF path with the spike-train filter to sense wideband channel. Note that *ISLA* does not need to meet tight timing constraints to switch since each subframe lasts 1 ms and there are multiple such subframes in each packet that can be leveraged for channel estimation. Thus, *ISLA* can simply skip a subframe while switching.

However, because *ISLA* captures the narrowband channel and wideband channel from different subframes, there is going to be an additional phase accumulation between the two measurements due to residual CFO. To address this, we slightly modify Eq.6, and the detailed description for this modification is presented in Appendix C.
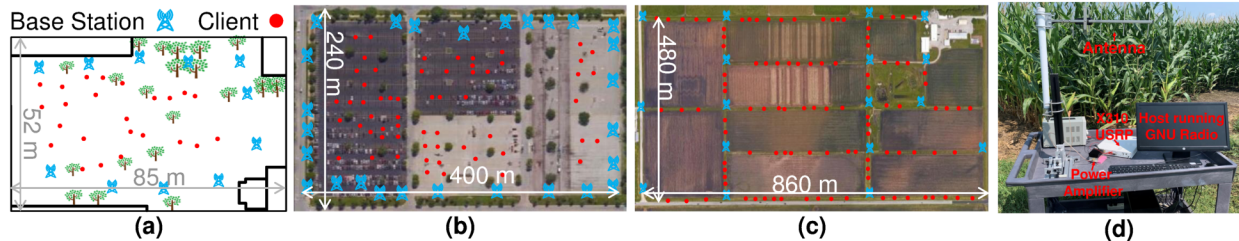
Figure 6: Ourdoor Experiment Testbeds: (a) Campus testbed surrounded by buildings. (b) Parking lot testbed. (c) Agricultural farm testbed. (d) Prototype base station in the agricultural farm testbed.

9 System Implementation

System Design: We have built a prototype *ISLA* device by combining our MEMS spike-train filter with commodity, off-the-shelf, low-power components. Figure 7(a) shows the circuit diagram, and Fig. 7(b) shows the actually prototype. It receives ambient 5G transmissions with two antennas followed by identical RF chains. Depending on whether the IoT devices wants to receive the full 100 MHz spectrum using the spike-train filter or the narrowband spectrum, the RF chains can switch between two paths: (1) the received wideband spectrum first be filtered by the MEMS spike-train filter, and then down-converted and sampled without using the anti-aliasing filter. (2) the MEMS spike-train filter is bypassed but the down-converted signal will first go through an anti-aliasing filter before sampling. We select between the two paths using RF switches controlled by a single microcontroller.

Implementation: We fabricated a MEMS spike-train filter at 400 MHz center frequency. However, due to the strong interference from the amateur radios in this band, we were not able to run experiments outdoor using this filter. Hence, the above prototype was only used indoors. In the outdoor experiments, we transmitted in a vacant 100 MHz wide spectrum between 950 and 1050 MHz, and we emulate the IoT radio front-end described above with the MEMS spike-train filters in digital using an X310 USRP software-defined radio (SDR). We would like to note that in practical deployments we do not expect interference to play a major issue since *ISLA* will be deployed in the proprietary frequency bands licensed by cellular companies, which in turn will have limited interference.

The X310 SDR has two identical RF chains, and can sample the full 100 MHz bandwidth with UBX160 daughterboards. To emulate the MEMS spike-train in digital, we first measure the spike-train filter frequency response once using a vector network analyzer (VNA), and we apply this filter frequency response to the received signals sampled at 100 MHz. Then, we downsample the filtered signal by simply keeping every 16th sample. This is equivalent to filtering the RF signal in analog and sample it below the Nyquist sampling rate. We also used a bandpass filters between the antenna and SDRs to remove out-of-band interferences and synchronized the two RF chains in time and phase through the GNU Radio Python API. In section 10.3, we present mircobenchmarks demonstrating the equivalence between applying the filter in
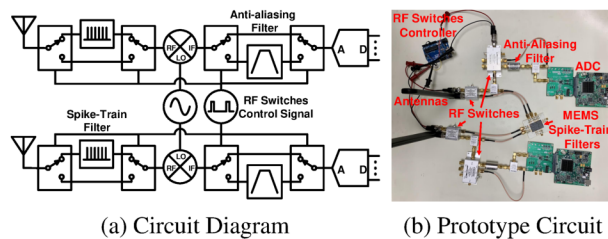

(a) Circuit Diagram   (b) Prototype Circuit

Figure 7: *ISLA* Prototype Circuit

digital and the above hardware prototype.

Testbed: Additionally, we also built 5G base station TX prototypes to transmit ambient 5G communication signals. As shown in Fig 6(d), the base station prototype consists an X310 USRP SDR with a UBX160 daughterboard, a 9 dBi Yagi directional antenna, and an RF Bay MPA-22-30 30 dB power amplifier. The base stations transmit 100 MHz OFDM packets. Using five base station prototypes, we created three testbeds with different dimensions and at different locations to conduct our experiments. Figure 6 shows the satellite images of our testbeds with the base stations and clients locations marked. The first testbed is 85 m long and 52 m wide on a university campus, surrounded by buildings on all sides. We designated 11 basestation locations in this testbed and chose five of them for each experiment. The second testbed is a 400 m by 240 m parking lot with 27 base station locations. The third testbed is at a 102 acre farmland with 860 m length 480 m width. We selected five out of the 17 potential locations to place the base stations in each experiment. For ground truth locations, we used differential GPS RTK with real-time RTCM correction data, which provides centimeter-level positioning accuracy.

10 Experimental Evaluation

10.1 Baselines

(1) *Spot-Fi:* [21] proposes a 2D MUSIC algorithm with spatial smoothing, which can localize clients by separating the multipath components jointly along the ToF and AoA domains.
(2) *mD-Track:* [43] separates propagation paths by leveraging multiple dimensions of the wireless signal (ToF, AoA, AoD and Doppler), and proposes an iterative algorithm that goes through multiple rounds of error computation and path re-estimation. In our experimental setup, leveraging the AoD and Doppler dimensions provides little benefit since the base
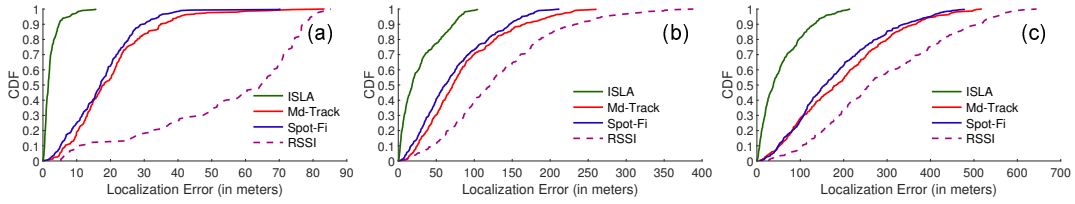
Figure 8: *ISLA*'s localization accuracy compared against baselines across different testbeds: (a) Campus (b) Parking lot (c) Farm.

station is equipped with a single antenna and the IoT device does not have high mobility relative to the base station. Note that, systems like Spot-Fi and mD-Track were not designed for ambient localization, and thus need to be adapted here. Specifically, we leverage the ToF estimates provided by these baselines for the LoS path, and in turn self-localize the client by computing the relative ToF, as described in Section 7. (3) *RSSI:* Past work leverages RSSI measurements to localize clients in outdoor cellular networks, by either using approximate path loss models for trilateration, or by using the known locations of nearby cells as coarse estimates. We implemented one recent RSSI baseline [9].

(4) *Spike-train filter-adapted baselines:* To provide a fair comparison against *ISLA*, we modify Spot-Fi and mD-Track to leverage the spike-train filter and utilize the wideband channel measurements for localization. It is non-trivial to adapt Spot-Fi for the spike-train filter since the spatial smoothing technique used in Spot-Fi requires uniformly spaced channel measurements across frequency, whereas the spike-train filter samples the OFDM frequency bins non-uniformly. To address this, we restructure the spatial smoothing subarray from [21] that allows Spot-Fi to be applied across the non-uniform frequencies sampled by the spike-train filter.

10.2 Results

Unless otherwise specified, for all results, we utilize 5 randomly chosen base stations as the anchor points.

A. Localization Accuracy Comparison against Baselines: We compare *ISLA*'s localization against the baselines in Fig. 8. Note that, while *ISLA* is designed specifically to leverage the wideband channel sensed by the MEMS filter, the baselines are implemented without modification and thus utilize only the narrowband channel for localization.

From Fig. 8, *ISLA* achieves a median localization accuracy of 1.58 meters in the campus testbed, 17.6 meters in the parking lot testbed, and 37.8 meters in the farm testbed. Across the same three testbeds, Spot-Fi achieves median accuracies of 17.05 meters, 61.2 meters and 156.6 meters, whereas mD-Track achieves 18.11 meters, 71.8 meters, and 183.1 meters respectively. Thus, *ISLA* improves the localization accuracy over Spot-Fi and mD-track by $\sim 11\times$ in the campus testbed, and by $\sim 4\times$ in the parking lot and farm. *ISLA* is able to achieve such high gains since it leverages the spike-train filter to sense wideband channel on the narrowband device, which allows for much higher resolution compared to the baselines

operating solely in the narrowband. Further, the localization improvement over the narrowband baselines is most significant in the campus testbed, since it has the most multipath from surrounding buildings, and thus ToF resolution is critical to separate out the LoS path from reflections.

Lastly, the RSSI baseline achieves median accuracies of 64.54 meters, 120.7 meters, and 260.8 meters respectively across the three testbeds. RSSI based methods generally have poor performance, as they tend to oversimplify path loss models that map RSSI values to distance, which does not hold for real world multipath channels.

B. Comparison against Spike-train-adapted Baselines: Next, we evaluate how leveraging the spike-train filter would benefit the performance of our narrowband baselines. Fig. 9 shows the CDF of localization accuracy comparing *ISLA* against the modified baselines that utilize the wideband channel from the spike-train filter. The RSSI baseline is not included here since its localization performance does not depend on bandwidth. Compared to its narrowband implementation, Spot-Fi's median accuracy improves to 11.08 meters in the Campus testbed, 49.07 meters in the Parking Lot, and 137.76 meters in the farm. Similarly, mD-Track's median performance improves to 15.48 meters, 51.45 meters and 103.78 meters in the three testbeds respectively. Thus, Spot-Fi and mD-Track see improvements in localization accuracy by up to 54% and 76% respectively. This shows that other localization techniques can also benefit from the wide-band channel sensing capabilities enabled by the spike-train filter.

Additionally, Fig. 9 shows that given the same channel information, *ISLA*'s off-grid CIR estimation algorithm is able to better resolve and estimate the relative ToF compared to Spot-Fi and mD-Track. This is because these baselines were designed to leverage multiple information dimensions to separate out the multipath components, with both baselines leveraging 3 or more antennas for separation in the AoA domain, and mD-Track further using the additional dimensions of Doppler and AoD as well. In contrast, here the IoT device has to separate out multipath in the ToF domain alone, and *ISLA* is able to achieve very accurate localization owing to its off-grid estimation algorithm.

C. *ISLA* Leveraging Different Amounts of Spectrum: In this experiment, we compare *ISLA*'s localization algorithm applied across three different amounts of spectrum utilization — (1) *ISLA* applied only to the wideband sparse channel sensed by the spike-train filter (without combining with narrowband channel), (2) *ISLA* applied only to the narrowband channel of
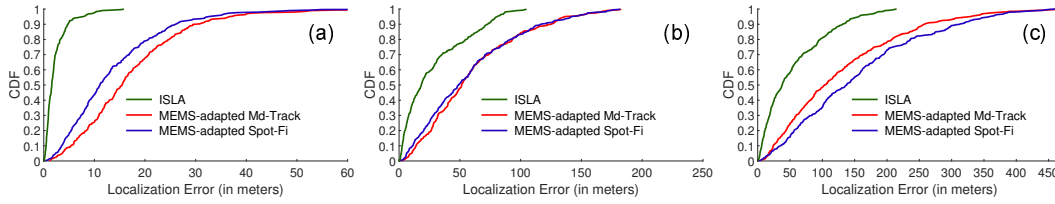
Figure 9: *ISLA*'s localization accuracy compared against MEMS filter adapted baselines at: (a) Campus (b) Parking lot (c) Farm.

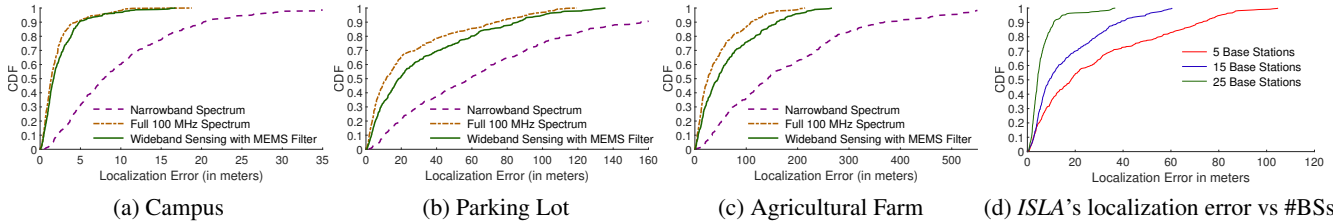| (a) Campus | (b) Parking Lot | (c) Agricultural Farm | (d) *ISLA*'s localization error vs #BSs |

Figure 10: (a-c) Comparison of *ISLA*'s localization accuracy when leveraging different amounts of spectrum across all three testbeds. (d) *ISLA*'s localization error with different number of visible base stations.

IoT device, and (3) *ISLA* applied across the entire 100 MHz bandwidth of the received 5G signal. Fig. 10 plots the CDF of localization accuracy achieved across the three testbeds.

ISLA applied on the narrowband channel performs the poorest, achieving median accuracies of 7.9 meters, 58.9 meters and 142.52 meters in the campus, parking lot and farm testbeds. In contrast, *ISLA* along with the spike-train filter can achieve corresponding median accuracies of 1.68 meters, 18.8 meters and 45.04 meters. Thus, *ISLA* along with spike-train achieves an improvement in localization accuracy of $3.16 \times -4.7 \times$ compared to *ISLA* applied in the narrowband spectrum, despite both baselines capturing the same amount of channel measurements. The advantage of spike-train stems from the fact that it enables the narrowband receiver to capture channel measurements that span a much larger bandwidth, which results in much higher ToF resolution.

On the other hand, *ISLA*'s localization algorithm applied on the full 100 MHz spectrum achieves median accuracies of 1.38 meters, 11.44 meters and 25.8 meters respectively on the three testbeds. Thus, *ISLA* with the spike-train filter reduces the localization accuracy by only $1.21 \times$, $1.64 \times$, and $1.74 \times$ respectively compared to this upper bound. This demonstrates that the spike-train filter can enable a narrowband device to achieve localization accuracy within a factor of $2 \times$ compared to a broadband receiver, despite the fact that it subsamples the signal by $16 \times$ below Nyquist.

D. Localization with Number of Anchor Base Stations:
In Fig. 10(d), we compare *ISLA*'s localization performance with 5, 15 and 25 base stations used as anchor points respectively, in the parking lot testbed. With 5 base stations, *ISLA* achieves a median accuracy of 17.6 meters, which improves to 9.27 meters with 15 base stations, and 4.26 meters with 25 base stations. This improvement becomes even more significant at the tail, with *ISLA* achieving 90th percentile accuracy of 73.16 meters with 5 base stations, which improves to 10.9 meters accuracy with 25 base stations at 90th percentile. Thus,

leveraging more base stations can significantly improve the localization accuracy achieved by *ISLA*.

E. Tracking Objects: We move the IoT device across an L-shaped trajectory (160 meters in length and 85 meters in width) in the parking lot testbed, and collect packet transmissions from the base stations at different points along this trajectory. In this experiment, we pick 7 fixed base stations to utilize as anchor points, and we show the ground truth trajectory and corresponding estimated trajectory by *ISLA* in Fig. 11(a). As can be observed, *ISLA*'s high localization accuracy allows to faithfully capture the shape of the ground truth trajectory.

10.3 Microbenchmarks

A. CIR Estimation using Fabricated MEMS Spike-train Filter: To verify the equivalence between our outdoor implementation and using the prototype with the fabricated MEMS spike-train filter at 400 MHz, we conduct indoor experiments at 400 MHz. Specifically, we evaluate the error in reconstructed CIR and estimated ToF values between the prototype with the fabricated filter and *ISLA* with the digital filter implementation. In Fig. 11(b), we show the CDF of the errors in ToF values (converted to distance (meters)) recovered by the two approaches, for both LoS and NLoS paths. We can see that the position of the LoS path in the CIR estimated from both approaches are very close, with the median error between their estimates being 0.075 meters. The error in the NLoS paths is higher, with a median error of 1.05 meters. However, this will not affect the localization performance between the two since localization only uses the LoS path. This microbenchmark demonstrates that *ISLA*'s approach of applying the filter and subsampling in digital is equivalent to using the fabricated filter from a localization perspective, and that the results shown in this paper are representative of a fully implemented system.
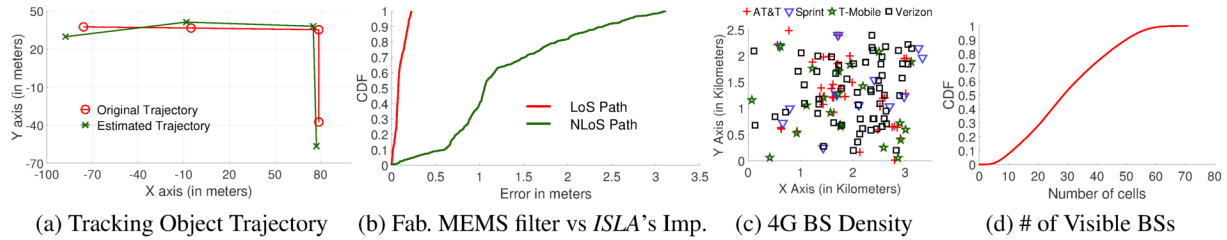
(a) Tracking Object Trajectory   (b) Fab. MEMS filter vs *ISLA*'s Imp.   (c) 4G BS Density   (d) # of Visible BSs

Figure 11: (a) Using *ISLA* to track object trajectory. (b) ToF difference between *ISLA*'s prototype with fabricated MEMS filter and digitally implemented MEMS filter. (c) Deployment of 4G base stations in the downtown area of a major US city. (d) Number of visible 4G base stations at various downtown locations.

Direction	NW	NE	SE	SW
Median	1.3535 m	1.3544 m	1.3267 m	1.3681 m
Std Dev	0.4948 m	0.6026 m	0.4908 m	0.512 m

Table 1: Invariance of Localization Error to Orientation

B. Density of Deployed Base Stations: In Section 10.2D, we have shown that *ISLA*'s localization accuracy increases substantially as we use more anchor base stations. Here, we study the distribution of how many base stations can the client overhear at a given location. Using publicly available databases [1], we retrieved the locations of 4G LTE base stations belonging to 4 major carriers in the United States. We chose 4G LTE for this analysis since 5G deployment is still in its nascent stage in the USA, but we expect the target coverage for 5G networks to exceed the 4G deployment.

In Fig. 11(c), we show the scatter plot of the 4G base stations located in the downtown area of a major metropolitan city in the USA. Using the cell coverage information provided in [1] for the different base stations, in Fig. 11(d), we plot the CDF of the number of base stations that the client can overhear at different locations on the map. We can see that at the 10th percentile, the number of visible base stations is 11, thus implying that less than 10% of client locations see less than 11 base stations. Further, the median number of base stations visible to the client is 29. This demonstrates that the cellular deployment is dense enough to allow many anchor points, which in turn can achieves high localization accuracy.

C. Invariance to Orientation: Here, we demonstrate that the localization performance is independent of the orientation of the IoT device. This is because the arcs that define the locus of the IoT node, depend only on the angle subtended by the base stations at the IoT device's location, which is invariant to device rotation. At a given location in our campus testbed, we orient the IoT device along 4 different directions and perform 100 localization experiments at each orientation. From Table 1, we can see that the median and standard deviation in localization error is almost the same across the 4 orientations, thus demonstrating invariance to orientation.

11 Limitations and Discussion

- *Power Footprint:* To enable ambient localization, *ISLA* leverages a second antenna and RF chain, which increases the power footprint of the IoT device. However, we would like to note that the power overhead of an additional RF chain is going to be lower than that of a GPS module, which is the likely alternative for localization. This is because the additional RF chain on the IoT device is going to operate in the narrowband with very low sampling rates, whereas GPS incurs high operational power since it needs to receive and correlate long sequences to get the signal power above the noise floor for GPS lock acquisition. Hence, while *ISLA*'s design does lead to an increased power footprint, it is still a better alternative compared to GPS.

- *Loss of SNR:* Since the MEMS spike-train filter is a passive device, the signal suffers from insertion loss when passed through the filter, thus resulting in loss of SNR. This is further exacerbated by the fact that in practice, the out-of-band rejection of the spike train filter is finite, which results in further loss of SNR. It is possible to reduce the impact of this SNR loss at the circuit level by improving impedance matching and the isolation between input and output ports. We can also compensate for the SNR loss by averaging the channel measurements across multiple OFDM symbols.

- *Line-of-sight*: Similar to many localization systems, *ISLA* assumes the availability of line-of-sight (LoS) paths to the base stations which might not hold under occlusion. This, however, can be addressed by potentially selecting a subset of base stations with LoS paths using similar techniques demonstrated in [21]. With the dense deployment of 5G base stations, we expect a significant subset of base stations to have LoS path to the node.

- *Fast Mobility*: The current design of *ISLA* is not suitable for highly dynamic applications with fast mobility such as tracking cars. This is because the localization algorithm must receive wideband 5G packets from 4 or more base stations before it can self-localize.

- *Multiple Providers*: *ISLA* can benefit from capturing signals from multiple different providers since the IoT node does not need to associate with the base stations. However, different providers operate in different frequency bands which would require different spike-train filters. This could potentially be addressed by having multiple filters and switching between them similar to our design in sec. 9.

References

[1] Cell Mapper cell tower locations. https://www.cellmapper.net. Accessed: Mon, Sep 13, 2021.

[2] 3GPP. Study on narrow-band internet of things (NB-IoT) / enhanced machine type communication (eMTC) support for non-terrestrial networks (NTN). Technical Report (TR) 36.763, 3rd Generation Partnership Project (3GPP), 06 2021.

[3] Godfrey Anuga Akpakwu, Bruno J Silva, Gerhard P Hancke, and Adnan M Abu-Mahfouz. A survey on 5g networks for the internet of things: Communication technologies and challenges. *IEEE access*, 6:3619–3647, 2017.

[4] Heba Aly and Moustafa Youssef. Dejavu: an accurate energy-efficient outdoor localization system. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 154–163, 2013.

[5] Atul Bansal, Akshay Gadre, Vaibhav Singh, Anthony Rowe, Bob Iannucci, and Swarun Kumar. Owll: Accurate lora localization using the tv whitespaces. In *Proceedings of the 20th International Conference on Information Processing in Sensor Networks (co-located with CPS-IoT Week 2021)*, pages 148–162, 2021.

[6] Sujittra Boonsriwai and Anya Apavatjrut. Indoor wifi localization on mobile devices. In *2013 10th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology*, pages 1–5. IEEE, 2013.

[7] Mathieu Bouet and Aldri L Dos Santos. Rfid tags: Positioning principles and localization techniques. In *2008 1st IFIP Wireless Days*, pages 1–5. IEEE, 2008.

[8] Stephen Boyd, Stephen P Boyd, and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[9] Rizanne Elbakly and Moustafa Youssef. Crescendo: An infrastructure-free ubiquitous cellular network-based localization system. In *2019 IEEE Wireless Communications and Networking Conference (WCNC)*, pages 1–6. IEEE, 2019.

[10] Sinan Gezici and Zafer Sahinoglu. Uwb geolocation techniques for ieee 802.15.4a personal area networks. *MERL Technical report*, 2004.

[11] Songbin Gong, Yong-Ha Song, Tomas Manzaneque, Ruochen Lu, Yansong Yang, and Ali Kourani. Lithium

niobate mems devices and subsystems for radio frequency signal processing. In *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 45–48. IEEE, 2017.

[12] Junfeng Guan, Jitian Zhang, Ruochen Lu, Hyungjoo Seo, Jin Zhou, Songbin Gong, and Haitham Hassanieh. Efficient wideband spectrum sensing using MEMS acoustic resonators. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 809–825. USENIX Association, April 2021.

[13] Fredrik Gustafsson and Fredrik Gunnarsson. Mobile positioning using wireless networks: possibilities and fundamental limitations based on available wireless network measurements. *IEEE Signal processing magazine*, 22(4):41–53, 2005.

[14] Ismail Guvenc and Chia-Chin Chong. A survey on toa based wireless localization and nlos mitigation techniques. *IEEE Communications Surveys & Tutorials*, 11(3):107–124, 2009.

[15] Yixue Hao, Min Chen, Long Hu, Jeungeun Song, Mojca Volk, and Iztok Humar. Wireless fractal ultra-dense cellular networks. *Sensors*, 17(4):841, 2017.

[16] Haitham Hassanieh, Lixin Shi, Omid Abari, Ezzeldin Hamed, and Dina Katabi. Ghz-wide sensing and decoding using the sparse fourier transform. In *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*, pages 2256–2264, 2014.

[17] Mohamed Ibrahim and Moustafa Youssef. Cellsense: An accurate energy-efficient gsm positioning system. *IEEE Transactions on Vehicular Technology*, 61(1):286–296, 2011.

[18] Mohamed Ibrahim and Moustafa Youssef. A hidden markov model for localization using low-end gsm cell phones. In *2011 IEEE International Conference on Communications (ICC)*, pages 1–5. IEEE, 2011.

[19] Vikram Iyer, Rajalakshmi Nandakumar, Anran Wang, Sawyer B. Fuller, and Shyamnath Gollakota. Living iot: A flying wireless platform on live insects. In *The 25th Annual International Conference on Mobile Computing and Networking*, MobiCom '19, 2019.

[20] Michio Kadota, Shuji Tanaka, Yasuhiro Kuratani, and Tetsuya Kimura. Ultrawide band ladder filter using SH0 plate wave in thin LiNbO3 plate and its application. In *2014 IEEE International Ultrasonics Symposium*, pages 2031–2034, 2014.

[21] Manikanta Kotaru, Kiran Joshi, Dinesh Bharadia, and Sachin Katti. Spotfi: Decimeter level localization using wifi. In *Proceedings of the 2015 ACM Conference on*

Special Interest Group on Data Communication, pages 269–282, 2015.

[22] Somansh Kumar and Ashish Jasuja. Air quality monitoring system based on IoT using raspberry pi. In *2017 International Conference on Computing, Communication and Automation (ICCCA)*, pages 1341–1346. IEEE, 2017.

[23] Ruochen Lu, Tomás Manzaneque, Yansong Yang, Jin Zhou, Haitham Hassanieh, and Songbin Gong. Rf filters with periodic passbands for sparse fourier transform-based spectrum sensing. *Journal of Microelectromechanical Systems*, 27(5):931–944, 2018.

[24] E Manavalan and K Jayakrishna. A review of internet of things (IoT) embedded sustainable supply chain for industry 4.0 requirements. *Computers & Industrial Engineering*, 127:925–953, 2019.

[25] Andreas Marcaletti, Maurizio Rea, Domenico Giustiniano, Vincent Lenders, and Aymen Fakhreddine. Filtering noisy 802.11 time-of-flight ranging measurements. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 13–20, 2014.

[26] Saman Naderiparizi, Yi Zhao, James Youngquist, Alanson P Sample, and Joshua R Smith. Self-localizing battery-free cameras. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, pages 445–449, 2015.

[27] Rajalakshmi Nandakumar, Vikram Iyer, and Shyamnath Gollakota. 3d localization for sub-centimeter sized devices. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, pages 108–119, 2018.

[28] Aymen Omri, Mohammed Shaqfeh, Abdelmohsen Ali, and Hussein Alnuweiri. Synchronization procedure in 5g nr systems. *IEEE Access*, 7:41286–41295, 2019.

[29] Jeongyeup Paek, Kyu-Han Kim, Jatinder P Singh, and Ramesh Govindan. Energy-efficient positioning for smartphones using cell-id sequence matching. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 293–306, 2011.

[30] Joan Palacios, Guillermo Bielsa, Paolo Casari, and Joerg Widmer. Communication-driven localization and mapping for millimeter wave networks. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 2402–2410. IEEE, 2018.

[31] Joan Palacios, Paolo Casari, and Joerg Widmer. Jade: Zero-knowledge device localization and environment mapping for millimeter wave systems. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2017.

[32] Anshul Rai, Krishna Kant Chintalapudi, Venkata N Padmanabhan, and Rijurekha Sen. Zee: Zero-effort crowdsourcing for indoor localization. In *Proceedings of the 18th annual international conference on Mobile computing and networking*, pages 293–304, 2012.

[33] Hamada Rizk, Ahmed Shokry, and Moustafa Youssef. Effectiveness of data augmentation in cellular-based localization using deep learning. In *2019 IEEE Wireless Communications and Networking Conference (WCNC)*, pages 1–6. IEEE, 2019.

[34] Hazem Sallouha, Alessandro Chiumento, and Sofie Pollin. Localization in long-range ultra narrow band IoT networks using rssi. In *2017 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, 2017.

[35] Ahmed Shokry, Marwan Torki, and Moustafa Youssef. Deeploc: a ubiquitous accurate and low-overhead outdoor cellular localization system. In *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 339–348, 2018.

[36] Y. Song and S. Gong. Wideband spurious-free lithium niobate rf-mems filters. *Journal of Microelectromechanical Systems*, 26(4):820–828, 2017.

[37] Parvathanathan Subrahmanya and Amir Farajidana. 5g and beyond: Physical layer guiding principles and realization. *Journal of the Indian Institute of Science*, 100:263–279, 2020.

[38] Adam Thierer and Andrea Castillo. Projecting the growth and economic impact of the internet of things. *George Mason University, Mercatus Center, June*, 15, 2015.

[39] Deepak Vasisht, Zerina Kapetanovic, Jongho Won, Xinxin Jin, Ranveer Chandra, Sudipta Sinha, Ashish Kapoor, Madhusudhan Sudarshan, and Sean Stratman. Farmbeats: An iot platform for data-driven agriculture. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 515–529, 2017.

[40] Deepak Vasisht, Swarun Kumar, and Dina Katabi. Decimeter-level localization with a single wifi access point. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 165–178, 2016.

[41] Jue Wang and Dina Katabi. Dude, where's my card? rfid positioning that works with multipath and non-line of sight. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, pages 51–62, 2013.

[42] Fuxi Wen, Henk Wymeersch, Bile Peng, Wee Peng Tay, Hing Cheung So, and Diange Yang. A survey on 5g massive mimo localization. *Digital Signal Processing*, 94:21–28, 2019.

[43] Yaxiong Xie, Jie Xiong, Mo Li, and Kyle Jamieson. md-track: Leveraging multi-dimensionality for passive indoor wi-fi tracking. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–16, 2019.

[44] Jie Xiong and Kyle Jamieson. Arraytrack: A fine-grained indoor location system. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 71–84, 2013.

[45] Jie Xiong, Karthikeyan Sundaresan, and Kyle Jamieson. Tonetrack: Leveraging frequency-agile radios for time-based indoor wireless localization. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, pages 537–549, 2015.

[46] Chouchang Yang and Huai-Rong Shao. Wifi-based indoor positioning. *IEEE Communications Magazine*, 53(3):150–157, 2015.

[47] Chaoyun Zhang, Paul Patras, and Hamed Haddadi. Deep learning in mobile and wireless networking: A survey. *IEEE Communications surveys & tutorials*, 21(3):2224–2287, 2019.

[48] C. Zuo, N. Sinha, and G. Piazza. Very high frequency channel-select mems filters based on self-coupled piezo-electric AlN contour-mode resonators. *Sensors and Actuators A: Physical*, 160(1):132 – 140, 2010.

A   Proofs

Here we re-state the lemmas and provide proofs.

Lemma 5.1 *For a sub-sampling factor P and N OFDM sub-carriers, the complex valued scaling factors for each subcarrier will be preserved upon aliasing if $N = z \times P$, for some integer z, given the aliasing results in no collisions.*

Proof of lemma 5.1: Assume that $x[n]$ is a discrete signal from 0 to $N-1$, and we are sub-sampling (or *decimating*) it by a factor of P, meaning $y[n] = X[n \times P]$ for some integer P. Then the Discrete Fourier Transform of $y[n]$, denoted by $\hat{Y}[k]$

is

$$
\begin{aligned}
\hat{Y}[k] &= \sum_{n=0}^{\lfloor N/P \rfloor - 1} x[nP] e^{-j2\frac{2\pi}{\lfloor N/P \rfloor} kn} \\
&= \frac{1}{P} \sum_{n=0}^{N-1} x[n] \sum_{m=0}^{P-1} e^{j\frac{2\pi}{P}mn} e^{-j2\frac{2\pi}{\lfloor N/P \rfloor}\frac{kn}{P}} \\
&= \frac{1}{P} \sum_{m=0}^{P-1} \Big(\sum_{n=0}^{N-1} x[n] e^{-j(\frac{2\pi}{N}n)(k\frac{N/P}{\lfloor N/P \rfloor} - \frac{N}{P}m)} \Big).
\end{aligned}
$$

Now if P divides N, in other words $N = Pz$ for some integer z, the above simplifies to

$$
\begin{aligned}
\hat{Y}[k] &= \frac{1}{P} \sum_{m=0}^{P-1} \Big(\sum_{n=0}^{N-1} x[n] e^{-j(\frac{2\pi}{N}n)(k-zm)} \Big) \\
&= \frac{1}{P} \sum_{m=0}^{P-1} \hat{X}[k - zm],
\end{aligned}
$$

where $\hat{X}$ is the DFT of $x[n]$. This proves that, as long as there is no collision, meaning that there is at most one index m in the above equation for which $\hat{X}[k-zm] \neq 0$, then the complex values of $\hat{X}[k]$ will be fully preserved upon sub-sampling. This proves the lemma.

We also point out that if P does not divide N, then the complex values are *not* preserved. Specifically, if N/P is not a proper integer, $\hat{Y}[k]$ will be in terms of $\hat{X}[k\frac{N/P}{\lfloor N/P \rfloor} - \frac{N}{P}m]$ where inside the argument, $k\frac{N/P}{\lfloor N/P \rfloor} - \frac{N}{P}m$, is not necessarily an integer. As a result, the original information of $\hat{X}[k]$ is never repeated in any of the $\hat{Y}$ indices. In fact, $\hat{Y}$ would closely relate to an interpolated version of $\hat{X}$ with the Dirichlet kernel.

Lemma 5.2 *Consider an OFDM symbol with N frequency subcarriers, indexed as $\{f_{\frac{-N}{2}}, \ldots, 0, \ldots, f_{\frac{N}{2}-1}\}$ with inter-frequency spacing of Δf, and a narrowband receiver that subsamples by $P\times$. If P^2 divides N, then the ideal filter parameters that meet all three requirements are: (1) $f_M^0 = f_{\frac{-N}{2}}$, (2) $\left(\frac{N}{P^2} - 1\right) \times \Delta f < \Delta S < \frac{N}{P^2} \times \Delta f$, and (3) $\Delta F = \frac{N}{P}(1 + \frac{1}{P}) \times \Delta f$.*

Proof of Lemma 5.2: First, we show that no two frequencies collide after aliasing. Let $q = \frac{N}{P}$, and assume that two frequencies f_α and f_β collide. Let f_α be k-th subcarrier (for $0 \leq k < P$) covered at the i-th passband ($0 \leq i < \left\lceil \frac{\Delta S}{\Delta f} \right\rceil$), and let f_β have k' and i' as corresponding indices. To collide after aliasing, $f_\alpha - f_\beta = (k - k')\Delta F + (i - i')\Delta f$ must be an integer multiple of $q\Delta f$. However, $|k - k'| \leq P - 1$ and $|i - i'| < \frac{N}{P^2}$. Thus $\frac{|f_\alpha - f_\beta|}{\Delta f} < \left(\frac{P-1}{P} + \frac{1}{P}\right)q = q$, meaning we must have $f_\alpha - f_\beta = 0$, proving the first design requirement. Second, we note that P passbands that do not overlap (since $\Delta S < \Delta F$), and each passband covers exactly $\frac{N}{P^2}$ subcarriers. We therefore have
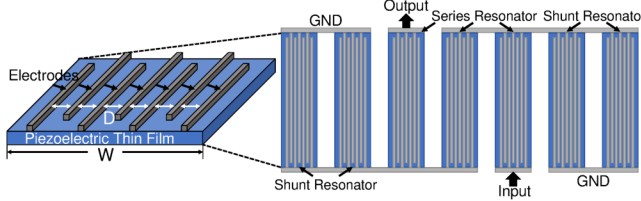
Figure 12: MEMS Spike-Train Filter Architecture

a total of $P \times \frac{N}{P^2} = q$ subcarriers that, as we just showed, do not overlap after aliasing. Therefore, after aliasing, each of the q subcarriers is covered exactly once, ensuring the second design requirement. Finally, we note that the smallest bin index is covered by the filter is $\min f_M = \frac{-N}{2}$, and the largest bin index is the last bin of the last passband, whose index can be computed as follows:

$$
\begin{aligned}
\max f_M &= \frac{-N}{2} + (P-1) \times \Delta F + \left\lceil \frac{\Delta S}{\Delta f} \right\rceil - 1 \\
&= \frac{-N}{2} + (P-1) \times \frac{N}{P}(1 + \frac{1}{P}) + (\frac{N}{P^2}) - 1 \\
&= -\frac{N}{2} + N - 1 = \frac{N}{2} - 1.
\end{aligned}
$$

Thus, the entire bandwidth (including $f_{\frac{-N}{2}}$ and $f_{\frac{N}{2}-1}$) is covered, ensuring the last design requirement.

B MEMS Spike-Train Filter

Spike-Train Filter Implementation: Following Lemma 5.2, we can derive the desired frequency response of the spike-train filter, and design MEMS resonators topology accordingly. For example, in our experiment, we used a 100 MHz 5G-like OFDM waveform with N=2048 subcarriers and a subcarrier spacing $\Delta f = 49\ kHz$, and we down-sample the filtered waveform by a factor of P=16. According to Lemma 5.2, the desired filter should 16 spikes with a spike spacing of 6.64 MHz spanning the 100 MHz bandwidth, and each spike should have a width around 400 kHz.

We can design a spike-train filter leveraging the periodic resonance frequencies of a type of MEMS acoustic resonators that is commonly referred to as a LOBAR (Lateral Overtone Bulk Acoustic Resonator). As shown in Fig. 12, the LOBAR resonator consists of 12 electrodes on the top of a thin film made of the piezoelectric material $LiNbO_3$. And we combine seven resonators in a ladder filter topology [20] to build a filter circuit. As a result, the LOBAR resonator architecture

determines the spike frequencies, whereas the slight difference between different resonators determines the width of the spikes. For simplicity, here we only focus on these two key parameters of the spike-train filter response, since they are restricted by our channel recovery algorithm as described in Sec. 5. More details on the MEMS spike-train filter design can be found in [23].

(1) The width of the film: the spacing between spikes Δf is determined by the width of the thin film W as $\Delta f = v/W$, where v is the acoustic velocity in the piezoelectric material, which is $\sim 4\ km/s$ in our design. Therefore, to achieve the 6.6 MHz spike spacing, we design the film width W to be $\sim 660\ \mu m$.

(2) The film width difference between different shunt and series resonators: the spike width ΔF of the spike-train filter equals to the resonant frequency difference between shunt and series resonators in the ladder filter, which is determined by the difference ΔW between shunt and series resonators: $\Delta F = f c \frac{\Delta W}{W}$. We design with piezoelectric film width to be $660\ \mu m$ for series resonators and $660.26\ \mu m$ for shunt resonators, which leads to $\Delta W = 0.26\mu m$, so that the widths of the spikes are around 400 kHz.

C Updated Objective Function to Account for Residual CFO

ISLA captures the narrowband channel and wideband channel from different subframes. Thus, there is going to be an additional phase accumulation between the two measurements due to residual CFO. To address this, we slightly modify Eq.6 where we split the objective function into two separate L-2 norm minimizations, with the first term containing only the wideband channel h'_M, and the second term containing only the narrowband channel h'_{NB}. This objective function is given below:

$$
\begin{aligned}
\{\tau_l^*\}_{l=1}^L = \underset{\tau_1, \dots, \tau_L}{\arg\min} \Big(&\|h'_M - V_M F_N \Psi (V_M F_N \Psi)^\dagger h'_M\|^2 \\
&+ \|h'_{NB} - V_{NB} F_N \Psi (V_{NB} F_N \Psi)^\dagger h'_{NB}\|^2 \Big)
\end{aligned}
\tag{7}
$$

$$
s.t. \quad \tau_l \geq 0 \quad \forall\, l \in \{1, 2, \dots, L\}
$$

The modified objective function is now invariant to phase offsets between the two channels, and *ISLA* can solve this updated optimization using the same technique described in Sec. 6.

Accelerating Collective Communication in Data Parallel Training across Deep Learning Frameworks

Joshua Romero[1], Junqi Yin[2], Nouamane Laanait[2],[*] Bing Xie[2], M. Todd Young[2], Sean Treichler[1],
Vitalii Starchenko[2], Albina Borisevich[2], Alex Sergeev[3],[†] Michael Matheson[2]
[1]NVIDIA, Inc. [2]Oak Ridge National Laboratory [3]Carbon Robotics

Abstract

This work develops new techniques within Horovod, a generic communication library supporting data parallel training across deep learning frameworks. In particular, we improve the Horovod control plane by implementing a new coordination scheme that takes advantage of the characteristics of the typical data parallel training paradigm, namely the repeated execution of collectives on the gradients of a fixed set of tensors. Using a caching strategy, we execute Horovod's existing coordinator-worker logic only once during a typical training run, replacing it with a more efficient decentralized orchestration strategy using the cached data and a global intersection of a bitvector for the remaining training duration. Next, we introduce a feature for end users to explicitly group collective operations, enabling finer grained control over the communication buffer sizes. To evaluate our proposed strategies, we conduct experiments on a world-class supercomputer — Summit. We compare our proposals to Horovod's original design and observe $2\times$ performance improvement at a scale of 6000 GPUs; we also compare them against tf.distribute and torch.DDP and achieve 12% better and comparable performance, respectively, using up to 1536 GPUs; we compare our solution against BytePS in typical HPC settings and achieve about 20% better performance on a scale of 768 GPUs. Finally, we test our strategies on a scientific application (STEMDL) using up to 27,600 GPUs (the entire Summit) and show that we achieve a near-linear scaling of 0.93 with a sustained performance of 1.54 exaflops (with standard error +- 0.02) in FP16 precision.

1 Introduction

The recent successes of Deep Neural Networks (DNNs) have encouraged continued investment across industries and domain sciences. Ranging from the traditional AI (e.g., image processing, speech recognition), to pharmaceutical and biomedical sciences (e.g., drug discovery), and to fusion, combustion and nuclear energy (e.g., disruption predictor, nuclear power plant) [29–34], more and more applications are actively exploiting ever-larger DNNs for production use.

With the growing applications of ever-larger DNNs, data parallelism in DNN training faces unprecedented challenges when synchronizing gradients[1] throughout distributed training runs. Deep learning (DL) frameworks, such as PyTorch [5] and TensorFlow [7], can exploit data parallelism for DNN training. In such a training run, an application creates multiple replicas of a model and distributes the replicas among a group of accelerators (e.g., CPUs, GPUs, TPUs, etc). Each accelerator executes on a different portion of training data across a number of *iterations*; at each iteration, it performs forward/backward pass computations independently, but synchronizes gradients (typically via global averaging) among the accelerators before applying weight updates (§2.1). In particular, accelerators synchronize tensors (multi-dimensional arrays) of gradients for the same set of parameters to ensure a globally consistent state for the model replicas.

This work advances collective communication in data parallel training. We propose several enhancements to Horovod [3] [25], a generic communication library designed to be independent to the framework runtimes, enabling its use across numerous popular DL frameworks with the same underlying backend implementation. Our ideas were motivated by two observations on Horovod. First, we observed that Horovod's core design is not scalable (see Figure 3) as it relies on a coordinator-worker control plane to orchestrate collective operations. At larger scales, this design choice leads to the single coordinator becoming overwhelmed and leaves the application runtime dominated by the orchestration process. Second, we found that Horovod's buffering mechanism (*Tensor Fusion*) fails to reliably generate optimal buffer sizes for efficient network bandwidth utilization (§2.2).

---

[1]Centralized training (also called synchronous training) synchronizes gradients among accelerators; decentralized training (asynchronous training) synchronizes parameters [13] [14] [21]. This work optimizes centralized training and discusses gradient synchronization accordingly.

To address these inefficiencies, we improve the control plane with a new coordination scheme that takes advantage of characteristics of a typical data parallel training paradigm, namely the repeated execution of collectives on a fixed set of gradients (§2.1). Using a caching strategy, we execute Horovod's existing coordinator-worker logic only once during a training run, replacing it with a more efficient decentralized orchestration strategy using a globally intersected bitvector for the remaining training duration (§3.1). Moreover, we introduce a feature for end users to explicitly group collective operations within Horovod, enabling finer grained control over the communication buffer sizes used for reductions.

While the implementation details vary, most DL-based communication libraries use similar design principles to optimize the performance of gradient synchronization. First, these libraries will employ mechanisms to facilitate overlapping of gradient synchronization and backward pass. That is, rather than waiting for gradients of all parameters to be computed and then synchronizing them across accelerators altogether at once, gradients will be synchronized actively as they are computed during the backward pass. Second, rather than launching a synchronization operator (e.g., AllReduce) for each gradient individually, the libraries employ bucketing/packing/fusion strategies (e.g., torch.DDP [18], tf.distribute [6], Horovod) to aggregate the gradients of multiple parameters and execute AllReduce on larger communication buffers for improved bandwidth utilization.

The contributions described in this work are mainly enhancements specific to Horovod, overcoming inefficiencies in its framework-agnostic design and original coordinator-worker strategy. The framework native communication libraries, like tf.distribute and torch.DDP, are closely integrated within their respective frameworks with access to internal details. With access to these details, the implementation of well-organized and performant communication and similar advanced features like grouping are simpler in these libraries. While the implementation details in this paper are Horovod specific, the proposed grouping technique is generally applicable to any other collective communication libraries.

In particular, we summarize our contributions as follows:

1. We implement a light weight decentralized coordination strategy by utilizing a response cache to enable Horovod to reuse coordination-related information collected at application runtime, accelerating the orchestration process.

2. We enable grouping to provide end users with explicit controls over tensor fusion in Horovod.

3. Our developments are incorporated in Horovod and publicly available in Horovod v0.21.0.

4. We conduct experiments to evaluate our solution on a world-class supercomputer — Summit. The results show that: 1) our solution outperforms Horovod's existing strategies across scales consistently. 2) Compared to the framework native communication libraries such like tf.distribute and torch.DDP, we achieve comparable and/or better performance across scales

consistently. Compared to a PS (parameter server)-based communication library BytePS [24], we achieve 20% better performance using up to 768 GPUs. 3) we further evaluate our solution on a scale up to 27,600 GPUs (the entire Summit) and show that we achieve near-linear scaling of 0.93 with a sustained performance of 1.54 exaflops (with standard error +- 0.02) in FP16 precision.

2 Background and Motivation

2.1 Data Parallelism in DNN Training

For data parallelism in distributed DNN training, a typical application run usually executes an iterative learning algorithm (e.g., SGD) among a number of GPUs; each GPU works on an identical replica and the same set of parameters of a DNN model. Here, a parameter is the bias or weight of a DNN layer; the value of a parameter or the value of a parameter's gradient is a multi-dimensional array, referred to as a *tensor*. In the run, a training dataset is partitioned into one or more equal-sized *batches*; each batch is processed on a different GPU. After a run starts, the model replicas, parameters, and the data structures of tensors are all fixed and determined.

During an iteration, each GPU updates parameters of a model replica by the following computational procedure: 1. the forward pass to compute loss. 2. the backward pass to compute gradients of the parameters. 3. the optimization step to update the parameters. In order to ensure model replicas are updated identically and remain in a globally consistent state, the gradients between GPUs are synchronized via averaging before updating parameters; this is referred to as *centralized learning*. Decentralized learning [13] [14] [21] maintains local consistency based on communication graphs [2] and synchronizes parameters. Moreover, for both centralized and decentralized learning, GPUs synchronize the same set of parameters/gradients across iterations. In this work, we focus on centralized learning and discuss collective communication in gradient synchronization/reduction.

Observation (1). For a DNN training run on a DL framework, the model replicas and parameters are all fixed. Across iterations in the run, GPUs repeatedly synchronize the same set of tensors for parameters/gradients.

2.2 Communication Libraries for Gradient Synchronization

2.2.1 Framework-native Libraries

For data parallel training, the key communication operations that occurs are AllReduce operations which average gradients among GPUs. Within an iteration, the framework processes

---

[2]In decentralized learning, GPUs are structured into a communication graph (e.g., ring or torus); each GPU only synchronizes among its local neighbors on the graph.

on GPUs each generate a set of gradients during the backward pass that must be globally reduced before being used to update the model parameters.

DL frameworks typically use dependency graphs to schedule compute operations, the use of which may result in non-deterministic ordering of operations. This is because in general, the order of operations through the compute graph that satisfies all dependencies is not unique. As a result, the order of operations executed can vary across framework processes within a single iteration, or even between iterations on a single process. This leads to problems in handling gradient communication between processes, as the operations generating gradients may occur in varied orders across processes. If each framework process naively executes a blocking AllReduce on the gradients in the order they are produced locally, mismatches may arise leading to either deadlock or data corruption. A communication library for DL must be able to manage these non-deterministic ordering issues to ensure that AllReduce operations are executed between processes in a globally consistent order.

The framework-native communication libraries (e.g., tf.distribute and torch.DDP) are designed to be closely integrated within the framework and have direct access to internal details, such as the model definition and expected set of gradients to be produced each iteration. Access to this information enables these libraries to directly discern the communication required during an iteration and more easily implement a performant communication schedule. For example, torch.DDP is a wrapper around a model in PyTorch, and utilizes the information contained in the model about gradients to determine how to schedule AllReduce operations during an iteration. While access to this information can simplify the implementation of these communication libraries, it ties their implementations strictly to the frameworks they were designed to support.

2.2.2 Framework-agnostic Libraries

In contrast to the framework-native communication libraries, a framework-agnostic library avoids any reliance on internal framework details and makes communication scheduling decisions based on information deduced during runtime. This design choice enables the library to operate across numerous frameworks, but the lack of access to internal information presents unique challenges. This section discusses the design of Horovod, a framework-agnostic communication library.

Horovod is a generic communication library developed to execute collective communication in data parallel training on GPUs, CPUs and TPUs, and with support for various DL frameworks. It serves as a high-level communication library that leaves network routing details (e.g., network reordering) handled by lower-level libraries, such as MPI, etc. Without loss of generality, this section discusses how Horovod integrates with MPI and TensorFlow on GPUs. Assuming this scenario, a distributed training run has N identical model repli-
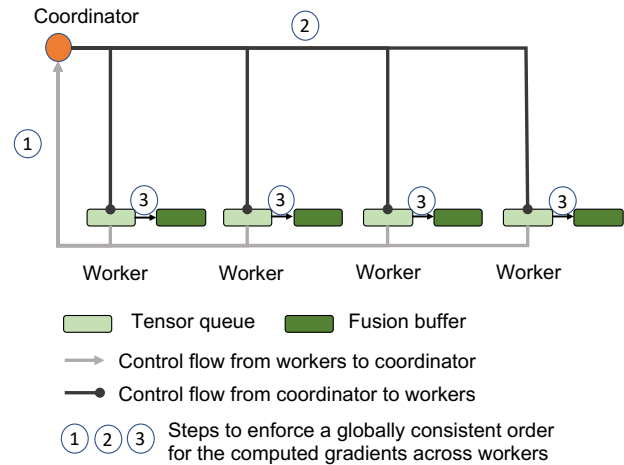

Figure 1: Coordinator-worker control model in Horovod's original design. The coordination progresses in three steps (see details in §2.2.2): First, the coordinator gathers the lists of requests from all workers; Second, the coordinator processes the request lists, and then generates and broadcasts a response list when observing one or more common requests from all workers; Third, after receiving the response list, each worker proceeds to execute collective operations.

cas, and is executed on N GPUs managed by Horovod with MPI and TensorFlow. In the run, each GPU serves as both an MPI rank and a TensorFlow process[3], which conducts computations for a model replica across iterations, with Horovod providing communication routines to synchronize gradients across TensorFlow processes.

This work introduces new techniques to Horovod after v0.15.2. In this section, we summarize the existing strategies based on v0.15.2. We use the terms *rank*, *process*, and GPU to refer to MPI rank, TensorFlow process, and their hosting GPU in turn, and use the terms *coordinator* and *worker* to refer to the Horovod threads spawned from the processes.

Similar to the framework-native libraries, Horovod must deal with the non-deterministic ordering of computations (discussed in §2.2.1). As it is agnostic to frameworks and lacking the knowledge of framework internal details, Horovod's design uses a control plane to resolve the non-deterministic ordering issue, where a coordinator-worker communication model is adopted to orchestrate collective communication and ensure a globally consistent order of execution.

Figure 1 presents a simple diagram of Horovod's control plane, with four threads each launched in a DL framework process. Particularly, the thread in Rank 0 serves as both the *coordinator* and a *worker*, and the other threads each serve as a different worker on a different GPU. During the course of a training run, the coordinator and workers execute the control logic periodically, with each execution referred

---

[3]For Horovod with TensorFlow, it is possible to use multi-GPUs per rank. But in production use, most users let each rank use a different GPU.

to as a *cycle*. In Horovod, the time between two sequential cycles is a configurable parameter with a default setting of 1 ms. To ensure synchronous cycles across Horovod threads, the communication operations in control plane (e.g., gather, broadcast) are blocking.

When a cycle starts, the coordinator first gathers lists of *requests* from all workers. Each request contains the metadata (e.g., tensor name, operation) that defines a specific collective operation on a specific tensor requested to be executed by the framework. The requests are collected from the worker's local tensor queue and are structured as a *request list*.

Next, the coordinator processes the request lists and counts the submissions of each request (identified by tensor name) from workers. When the coordinator observes that a common request has been submitted by all workers, it prepares that request for execution by generating a corresponding *response*. The coordinator generates a list of responses and broadcasts the list to all workers. Here, each response contains the metadata (e.g., tensor names, data type, collective operation) that is used by the Horovod backend to execute a collective operation (e.g., AllReduce). Optionally, before broadcasting, the coordinator will preprocess the response list, aggregating multiple compatible responses into larger *fused* responses, a process referred to as *Tensor Fusion* in Horovod documentation.

After receiving the response list, each worker proceeds to execute collective operations, one operation per response in the received response list. The portion of the Horovod backend executing collective operations is referred to as the *data plane*. For each response, a worker will access required input tensor data from the framework, execute the requested collective operation, and populate the output tensors for the framework's continued use. A key characteristic of this design is that the order of execution for collective operations is defined by the order of responses in the list produced by the coordinator. As such, a globally consistent ordering of collective operation execution is achieved across workers.

At a high-level, Horovod's design can be described as a set of mailboxes, where each worker is free to submit request for collectives in any order to their assigned mailbox, and eventually retrieve the desired output. The control plane is responsible for coordinating these requests across mailboxes, ensuring that only requests submitted by all workers are executed and are executed in a globally consistent order. One observation from this analogy is that Horovod's design is inherently unaware of any aspects of DL training, in particular that in typical DL workloads, a fixed set of gradient tensors will be repeatedly AllReduced during the course of a training run (discussed in §2.1). As a result, Horovod's design unnecessarily communicates redundant information to the coordinator at every iteration, leading to poor scalability.

Beyond coordination alone, tensor fusion may cause inefficiency in the data plane. Ideally, the tensor fusion process will generate well balanced fused responses throughout training, yielding larger sized communication buffers for improved
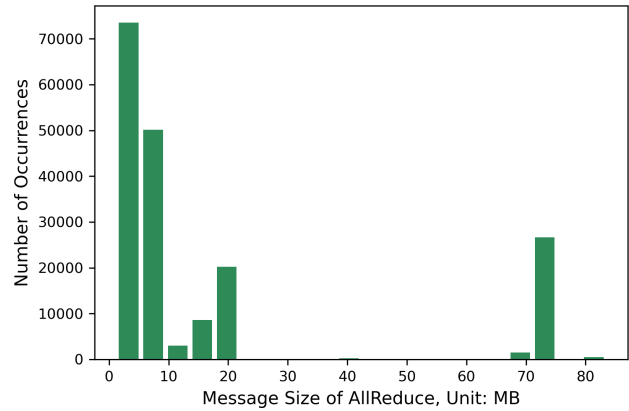

Figure 2: Histogram of AllReduce message size in Horovod's original design of Tensor Fusion. We present the results of a training run of ResNet50 with 96 GPUs on Summit (§4.1).

network utilization. In practice, as the tensor fusion is closely tied to cycle that runs at an arbitrary user-defined tic rate, the resulting communication buffer sizes can be highly dynamic and varied, even when comparing iteration to iteration in a run. Figure 2 presents the fused AllReduce message sizes on ResNet50 as an example to illustrate the performance of tensor fusion. In summary, it is possible to have the Horovod cycles occur at favorable times during the training iteration, where the collective responses are well distributed across the Horovod cycles running during the iteration, resulting in correspondingly well-balanced fused communication message sizes. On the other hand, the Horovod cycles can occur at unfavorable times during the iteration, with some cycles completing with a few or even just one available collective response, yielding less efficient communication on smaller buffers. In the worst case, a single trailing gradient tensor for the iteration can be missed by all previous cycles run during the iteration, inducing additional latency equal to the user-defined cycle time, just to reduce a single gradient tensor.

We report the detailed information about the original design of Horovod's control plane in the supplementary materials (Section 1), including pseudo code listings for Horovod coordinator-worker coordination logic and Horovod cycle, and the data structures for request list and response list.

Observation ②. The dynamic nature of tensor fusion can fail to generate buffer sizes for efficient network utilization. Thus, we are motivated to introduce a more explicit and strict control mechanism for tensor fusion that can improve performance.

2.2.3 Hierarchical Approach in Horovod

Kurth et al. [15] were the first to observe the scaling issue in Horovod's control plane. In particular, the coordinator-worker coordination strategy was found to be highly inefficient. When increasing the number of workers, the time cost of the communication and processing grows linearly since the coordinator needs to communicate/process the request list

from each worker. Especially at large scale, the cost of this co-ordination strategy was found to quickly dominate the training runtime. Their proposed solution was to introduce a hierar-chical tree-based variant of the original coordinator-worker control model, using a hierarchy of coordinators splitting up the coordination tasks. It is clear that this hierarchical con-trol strategy outperforms the original control plane with a logarithmic complexity, but at the same time, it suffers from the same issue as the original strategy does: the hierarchical coordination strategy redundantly communicates metadata for repeated operations across iterations in a training run.

Beyond the hierarchical coordination strategy, the authors also introduced a hybrid/hierarchical AllReduce in Horovod's data plane. Even with these improvements, their approach was not able to achieve efficient scaling with Horovod, requiring the introduction of a *gradient lag*. With gradient lag enabled, the gradients of a previous iteration are used to update weights in the current step, providing a longer window for overlapping the slower communication at scale with computation.

We present the hierarchical control plane in detail in the supplemental materials (Section 1) and discuss the perfor-mance of the hierarchical approach in Section 2.3.

Observation ③. Although existing Horovod solutions adopt different coordination strategies, they both fail to take ad-vantage of characteristics of DL workloads and repeat the same metadata communications in the control plane across iterations in a training run.

2.3 Discussions on Horovod Performance

We focus on understanding the performance of ex-isting Horovod solutions, including `Horovod_MPI`, `Horovod_NCCL`, and the hierarchical AllReduce (`Hierarchical_AllReduce`). Here, `Horovod_MPI` refers to the Horovod implementation with MPI for both the coordinator-worker communication in the control plane and AllReduce in the data plane. `Horovod_NCCL` refers to the im-plementation that uses MPI for control plane communication and NCCL for AllReduce in the data plane. In particular, NCCL v2.4.0 was used in this experiment, with tree-based communication algorithm options available along with existing systolic ring algorithm. `Hierarchical_AllReduce` represents the solution using MPI for the control plane communication and MPI+NCCL for the AllReduce in the data plane. In all three solutions, the coordinator-worker communication uses the control plane as shown in Figure 1. Moreover, all these solutions are available in Horovod [3].

We conducted experiments on STEMDL (See supplemen-tary materials Section 3), a scientific application developed to solve a long-standing inverse problem on scanning trans-mission electron micro-scopic (STEM) data by employing deep learning. The DNN model in STEMDL is a fully-convolutional dense neural network with 220 million parame-ters; each GPU generates/reduces 880MB of gradients at an

iteration. We ran the experiments on Summit supercomputer (§4.1), where each Summit node contains 6 GPUs.

We first consider the scalability results, shown in the left subfigure of Figure 3. It is clear that, after introducing the tree-based communication algorithms, `Horovod_NCCL` is able to deliver the best performance for all scales. When we in-crease the number of GPUs, `Horovod_NCCL` expands its lead in system throughput. For example, when using 6000 GPUs, it outperforms `Hierarchical_AllReduce` and `Horovod_MPI` by 3.2× and 5.4×, respectively.

Figure 3 (right subfigure) also reports the GPU utilization of the Horovod solutions across scales. The results show that, across all tested configurations, the GPU utilization is below 55%. When increasing the number of GPUs, the GPU uti-lization decreases progressively. We observed a much lower GPU utilization with 6000 GPUs (see Figure 6). This in-dicates that, although the NCCL-based AllReduce delivers good performance, the entire gradient reduction procedure in Horovod (e.g., coordination and execution) is highly ineffi-cient. It leaves GPU resources underutilized and compromises system throughput. In this work, we argue that the inefficiency originates from both the control plane and AllReduce and in-troduce techniques (discussed in §3) to overcome these issues.

We limit the evaluation on `Horovod_MPI` to 1536 GPUs as we see noticeably poor performance. We skip the evaluations of the hierarchical tree-based coordinator-worker communi-cation (Figure 1 in supplementary materials) and the gradient lag proposed in the hierarchical approach (§2.2.3), as they are currently neither included as part of Horovod nor publicly available. To summarize, Kurth et al. reported in [15] that, the entire hierarchical approach obtained the parallel efficiencies of ∼ 60% when using fully synchronous gradient reduction, only achieving above 90% on the Summit supercomputer with gradient lag enabled. In particular, researchers showed that gradient lag sometimes yields low training accuracy, and concluded that, without carefully tuning the related hyperpa-rameters, this type of techniques is not generally applicable to DNN training [9, 10, 20]. Moreover, we show that our solution obtains up to 93% of parallel efficiency on Summit using a fully synchronous gradient reduction (discussed in §4.4), 1.5× better than the performance of the hierarchical approach without gradient lag reported in [15].

3 Boosting Collective Communication in DNN Training with Caching and Grouping

This work proposes to advance collective communication in centralized learning across various DL frameworks. We introduce new techniques to Horovod to improve its scala-bility and efficiency in both the control plane and the data plane. For the control plane, we develop a strategy to record the coordination information on the repeated requests for the same collective operations on the same parameters across
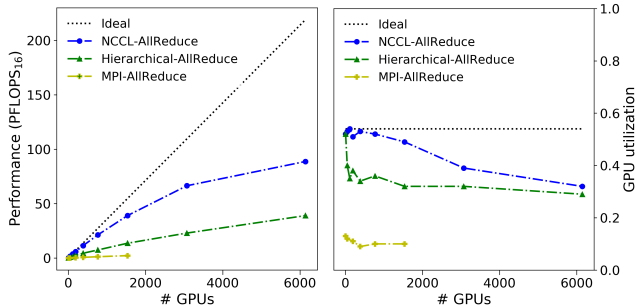
Figure 3: Performance and GPU utilization of existing Horovod strategies on STEMDL workload.

iterations in a training run (discussed in §2.1). In particular, we develop a light weight decentralized coordination strategy by utilizing a *response cache*. This cache introduces a means for Horovod to store the metadata about the repeated collective requests at each worker locally and bypass the redundant coordinator-worker communication entirely after the cache is populated. Moreover, we introduce *grouping* as a feature to Horovod's data plane. With grouping enabled, a user can request grouped collective operations for specific tensor groups, enforcing explicit control over Horovod's tensor fusion. We later show in experiments (§4) that, these two techniques can lead to significant performance improvement and obtain near-linear scaling in the production runs on a world-class supercomputer. Our techniques are adopted by Horovod and are publicly accessible in v.0.21.0.

In general, our proposals are built within Horovod's existing control logic (discussed in §2.2.2): we execute cycles to coordinate collective communication in DNN training; in our system, blocking communications are used to ensure synchronous cycles across workers and the network routing details (e.g., network reordering) are managed by lower-level communication libraries, such as MPI. Additionally, our modifications support both MPI and Gloo [2] libraries for control plane communication. We discuss the performance evaluation using MPI for control plane communication and either MPI or NCCL for data plane communication in Section 4.

3.1 Orchestrating Collective Communication with Caching

In contrast to the framework-native communication libraries like tf.distribute or torch.DDP, Horovod is designed to be generic. It utilizes lightweight bindings into frameworks to allow the Horovod runtime to process gradient reduction, and has no access to any data associated with the framework runtimes (e.g., iteration, parameters, models, etc.). In particular, Horovod interacts with DL frameworks via custom framework operations that enable the frameworks to pass a tensor and requested collective operation to the Horovod backend, and receive the output tensor after the collective is executed. These

custom operations are defined for each supported framework, as the mechanisms to share tensor data can vary between frameworks, but otherwise the remainder of the code base is generic. This design choice enables Horovod to work across numerous DL frameworks using the same underlying code, but at the same time, this generic design leads to the inefficiency at scale with its centralized coordinator-worker control plane.

As is summarized in **Observation (1)**, in a typical data parallel training run, there is a fixed set of gradients that needs to be AllReduced across iterations. Horovod's existing coordinator-worker design does not take advantage of this aspect of the workload, and will redundantly process the same collective communication requests through the coordinator at each iteration (**Observation (3)**). Although this design choice allows Horovod to be dynamic and service any collective request submitted from workers, it is unnecessarily inefficient for the typical use case with a fixed set of repeated collective operations.

This section introduces a caching strategy that enables Horovod to capture and register repeated collective operations at runtime. With the cached metadata, we build a decentralized coordination among workers, replacing the existing strategy with significant performance improvement.

3.1.1 Response Cache

As Horovod does not have direct access to the framework-runtime metadata (e.g., iteration, tensors), any pattern of collective operations launched during a training run must be deduced at runtime based on prior collective requests observed. In order to capture the metadata about repeated collective operations, we introduce a *response cache* to Horovod. This cache can be used to identify repeated operations, as well as store associated *response* data structures generated by the coordinator to be reused without a repeated processing through the coordinator-worker process.

Each worker maintains a response cache locally. To construct the cache, Horovod threads will use the existing coordinator-worker control plane implementation. Specifically, workers send requests to the coordinator and receive a list of responses from the coordinator to execute. Instead of executing the collective operations immediately and destroying the response objects, the workers first store the response objects in a local cache, where each unique response is added to a linked-list structure. Additional tables are kept mapping tensor names to response objects in the cache as well as integer position indices in the linked list. A key characteristic of the cache design is that its structure is fully deterministic based on the order that response entries are added to the cache. In this design, the cache is populated using the list of responses received by the coordinator when a collective request is first processed. As the coordinator design already enforces a global ordering of responses, responses are added
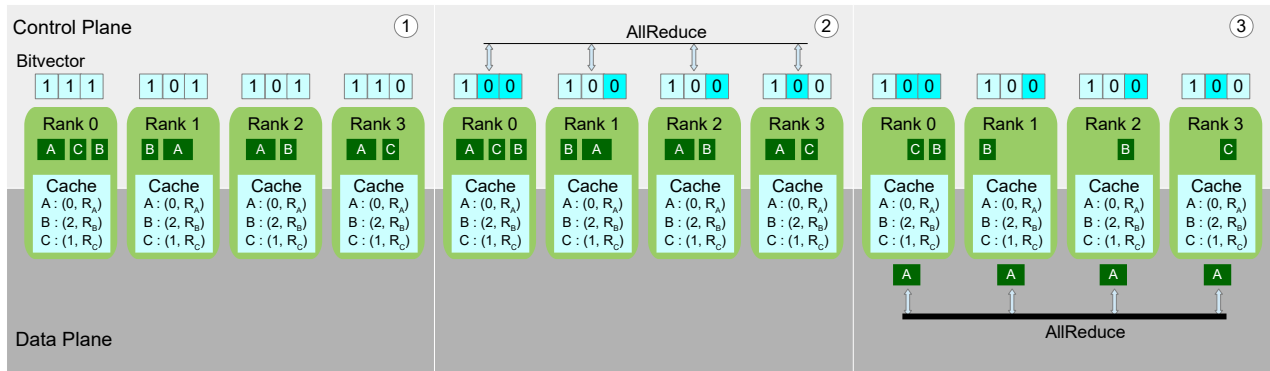
Figure 4: Illustration of AllReduce with Caching. We depict an example with 4 workers (0, 1, 2, 3) reducing 3 tensors (A, B, C). The strategy works in three steps: 1. Each worker populates a *bitvector*, setting bits according to entries in the *response cache* and the pending tensors in their local queues. 2. Workers synchronize the bitvectors via a global set intersection to identify common pending tensors. In this example, the bit associated with tensor A is shown as common across the workers. 3. Tensor A is sent to the data plane for AllReduce. When the AllReduce operation is done, Tensor A is removed from the queues on all workers.

to the cache on each worker in a globally consistent order which in turn ensures caches on each worker remain identical across workers. The data structure for each entry in the cache is the same as a response list discussed in Section 2.1. The cache implemented has a user-configurable capacity, with a default size of 1024 unique responses.

Using a combination of the cached responses and the globally consistent structure of the caches, a lightweight decentralized coordination scheme is enabled, as illustrated in Figure 4.

3.1.2 Cache-based Coordination with Response Cache and Bitvector

Once the response cache is created, it is utilized together with a bitvector to implement a lightweight decentralized coordination scheme. To achieve this, we take advantage of the fact that the response cache is constructed in a way that guarantees global consistency across workers. As a result, the structure of the response cache, in particular the index position of cached response entries, can be used to maintain a global indexing scheme of requests that are repeated that can be leveraged for coordination. We present the strategy in Figure 4, report the corresponding pseudo code in Algorithms 1 and 2, and summarize its procedure below.

1. At the start of a cycle, each worker performs the same operations as it does in the original design: it retrieves the pending requests from its local tensor queue, yielding a *RequestList*.

2. Each request in *RequestList* is checked against the response cache. If the request has an associated entry in the cache, the position of the cached entry is added to a set, *CacheBits*. Otherwise, this request does not have an associated cached entry and a flag is set to indicate that an uncached (i.e. previously unobserved) request is pending.

Algorithm 1 Horovod cycle with caching

1: **procedure** RunCycleOnce
2:      $RequestList \leftarrow$ PopMessagesFromQueue()

3:      $CacheBits_g, UncachedInQueue_g \leftarrow$ CacheCoordination($RequestList$)

4:      $UncachedRequestList \leftarrow []$
5:      **for** M in $RequestList$ **do**
6:          $cached \leftarrow ResponseCache$.cached($M$)
7:          **if** $cached$ **then**
8:              $bit \leftarrow ResponseCache$.GetCacheBit($M$)
9:              **if** $bit \notin CacheBits_g$ **then**
10:                  PushMessageToQueue(M)   ▷ Replace messages corresponding to uncommon bit positions to framework queue for next cycle

11:              **end if**
12:          **else**
13:              $UncachedRequestList$.append(M)   ▷ Collect any uncached messages
14:          **end if**
15:      **end for**

16:      $ResponseList \leftarrow ResponseCache$.GetResponses($CacheBits_g$)   ▷ Retrieve cached responses corresponding to common bit positions

17:      **if not** $UncachedInQueue_g$ **then**   ▷ All messages cached, skip master-worker coordination phase
18:          $FusedResponseList \leftarrow$ FuseResponses($ResponseList$)   ▷ Tensor Fusion
19:      **else**   ▷ Use master-worker coordination to handle uncached messages
20:          $FusedResponseList \leftarrow$ MasterWorkerCoordination($UncachedRequestList$, $ResponseList$)
21:      **end if**

22:      **for** R in $FusedReponseList$ **do**
23:          $ResponseCache$.put(R)   ▷ Add response to cache
24:          PerformOperation(R)   ▷ Perform collective operation
25:      **end for**
26: **end procedure**

---

3. Each worker populates a bit vector, *BitVector*, setting bits corresponding to values in *CacheBits*. It also sets a bit to indicate whether it has uncached requests in its queue. The bit vectors across workers are globally intersected using an

---

Algorithm 2 Decentralized coordination with response cache and bitvector

```
 1: procedure CACHECOORDINATION(RequestList)
 2:     CacheBits ← {}, UncachedInQueue ← False

 3:     for M in RequestList do              ▷ Check for cached messages
 4:         cached ← ResponseCache.cached(M)
 5:         if cached then
 6:             bit ← ResponseCache.GetCacheBit(M)
 7:             CacheBits.insert(bit)        ▷ Collect bit positions for
                                                cached entries
 8:         else
 9:             UncachedInQueue ← True       ▷ Record uncached message
                                                exists
10:         end if
11:     end for

12:     BitVector ← SetBitvector(CacheBits, UncachedInQueue) ▷ Set bits in local
        bitvector
13:     BitVector_g ← Intersect(BitVector)   ▷ AllReduce using binary
                                                AND op to get global
                                                bitvector
14:     CacheBits_g, UncachedInQueue_g ← DecodeBitVector(BitVector_g)   ▷ Get
        common bit positions and flag

15:     return CacheBits_g, UncachedInQueue_g
16: end procedure
```


Figure 5: Illustration of Grouping. A task graph with nodes that generate requests T_n is depicted on the left, with the dashed boxes indicating requests visible to `Horovod` at 3 subsequent cycles. The nodes are colored to depict assignment to two groups (blue/solid borders and green/dashed borders). By default, a worker will submit all requests observed in a cycle to be processed/executed which can yield unbalanced sets of requests. With grouping enforced, requests are only submitted when complete groups are available.

AllReduce with the binary AND operation, resulting in a globally reduced bitvector, $BitVector_g$. Through this operation, only bits corresponding to requests that are pending on all workers remain set, while others are zero.

4. Each worker decodes $BitVector_g$, collecting indices of any remaining set bits to form $CacheBits_g$, the set of cache indices corresponding to requests currently pending on all workers. Additionally, it extracts whether any worker has pending uncached requests in queue.

5. Each request in $RequestList$ is checked against the entries in $CacheBits_g$. If the request has an associated cache entry but has a position not in $CacheBits_g$, this means that only a subset of workers have this cached request pending. This request is pushed back into the internal tensor queue to be checked again on a subsequent cycle. If the request has an associated cache entry with a position in $CacheBits_g$, this means that the request is pending on all workers and is ready for communication. The associated response is retrieved from the cache and added to the $ResponseList$. If the request is not cached, it is added to a list of uncached requests that needs to be handled via the coordinator-worker process.

6. If there are no uncached requests pending on any worker, the coordinator-worker process is completely skipped and workers proceed to process locally generated $ResponseLists$ composed of response entries from the cache. Otherwise, uncached requests are handled via the coordinator-worker process, with the coordinator rank generating a $ResponseList$ containing the cached response entries along with new responses corresponding to the uncached requests.

It is worth highlighting that with this cache-based control, the coordinator-worker logic is only executed during cycles where previously unobserved requests are submitted to Horovod. In cycles where all requests are cached (i.e. repeated), the coordinator-worker control plane is never exe-
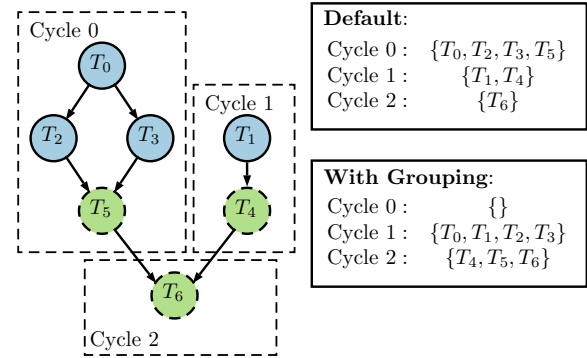
cuted. For a typical DL workload with a fixed set of gradients to reduce every iteration, the response cache will eventually contain entries corresponding to this entire set. As a result, the poorly scaling coordinator-worker process will be skipped for all training iterations, except the first one, where all requests are initially observed and placed into the cache.

3.2 Grouping

The response cache described in the previous section addresses inefficiencies in the Horovod control plane. In this section, we describe a method to improve the data plane performance of Horovod through explicit grouping of AllReduce operations. In particular, we introduce a feature to Horovod that enables users to submit grouped collective operations, allowing explicit control over Horovod's tensor fusion (§2.2.2).

As is shown in Figure 5, in place of submitting individual collective requests per tensor, a user can submit a grouped collective (e.g. `hvd.grouped_allreduce`) for multiple tensors. Collective requests submitted within a group are treated as a single request in Horovod's control plane; that is, no request in the group is considered ready for the data plane until all requests in the group are submitted. As a result, the tensors within a group are guaranteed to be processed by the data plane during the same cycle and fused, along with any other responses ready for execution during the cycle.

This new grouping mechanism can be used to control how gradient AllReduces are scheduled during an iteration. In particular, the gradient AllReduce requests for a single iteration can be assigned to one or more groups to explicitly control the fused communication buffer sizes that Horovod

generates for gradient reduction, avoiding issues that can arise using the default dynamic fusing strategy as described in Section 2.2.2. To ease use, this functionality is exposed to users via a new argument, `num_groups` to Horovod's high-level `DistributedOptimizer` wrapper. By setting this argument, the set of gradient tensors to be AllReduced within the iteration are evenly distributed into the number of groups specified. In the implementation described here, the gradients lists are split into groups of equal number of tensors, without consideration of buffer size.

Beyond this basic splitting, advanced users can achieve more optimal data plane communication performance by manually tuning the distribution of gradient tensors across the groups, to target fusion buffer sizes for improved network efficiency and/or achieving better overlap of communication and computation. We discuss the performance with different grouping configurations in Section 4.

We note that the framework native communication libraries like torch.DDP also support gradient fusion/bucketing and expose options to split gradient reduction into groups of approximately fixed message size. These native implementations generally leverage access to framework-level details, like information about the constructed model, to form these groups. As Horovod does not have access to these framework-level details directly, this grouping mechanism provides a means to provide such information via associating sets of tensors coming from the model to groups.

4 Experiment

4.1 Environment Setup

Hardware. We performed all experiments on Summit supercomputer [27] at the Oak Ridge Leadership Computing Facility. As the 2nd fastest supercomputer in the world, Summit is a 148.6 petaFLOPS (double precision) IBM-built supercomputer, consisting of 4,608 AC922 compute nodes with each node equipped with 2 IBM POWER9 CPUs and 6 NVIDIA V100 GPUs. Summit is considered as ideally suited for Deep Learning workloads, due to its node-local NVMe (called burst buffer) and Tensor Cores on V100 for faster low-precision operations. Moreover, its NVLink 2.0 and EDR InfiniBand interconnect provides 50 GB/s and 23 GB/s peak network bandwidths for intra-node and inter-node communication.

Software. The techniques proposed in this work are implemented based off Horovod v0.15.2 and have been incorporated in v0.21.0. We measured the performance with two communication backends, including NCCL v2.7.8 and Spectrum MPI (a variant of OpenMPI) v10.3.1.2. To evaluate the performance of our proposals across DL frameworks and to compare against the state-of-the-art communication libraries, we integrated our solutions in Horovod with TensorFlow (v2.3.1) and PyTorch (v1.6.0). We compared our solutions to tf.distribute in TensorFlow v2.4 (TensorFlow supports grouping since
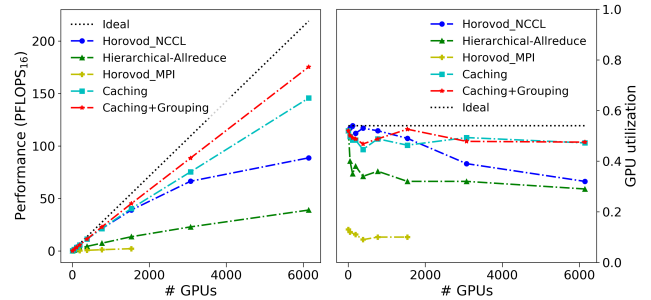

Figure 6: Performance and GPU utilization of Horovod's strategies. We compare our new techniques to the existing Horovod implementations using STEMDL (see Figure 3).

v2.4), torch.DDP in PyTorch v1.6.0, and BytePS (v0.2.5). In particular, BytePS is a deep learning framework that adopts PS (parameter server) as its communication model. BytePS is considered as an alternative to Horovod in a cloud environment. For tf.distribute and torch.DDP, we conducted the experiments with both NCCL and MPI; for BytePS, we conducted experiments simply with NCCL as BytePS does not support MPI. We configure BytePS in co-locate mode with one server and one worker per Summit node. We choose this configuration because it is recommended by the BytePS team as the best practice for high-performance computing (HPC) [1]. Moreover, we evaluated the scalability of our techniques with STEMDL, where the results are from an earlier incarnation of this work based on Horovod v0.15.2 built with NCCL v2.4, but the conclusions are similar.

Workloads. We evaluated our solution on GPU-based workloads. Starting with the STEMDL workload (message size 880MB per GPU), we compared our new techniques to the existing Horovod strategies (see Figures 3 and 6) with TensorFlow. We then broadened the experiments to compare with tf.distribute, torch.DDP, and BytePS on Resnet50 (102MB per GPU). Finally, we demonstrated our approach on ResNet50 and two more popular networks: EfficientNet-B0 (21MB per GPU) and VGG19 (574MB per GPU). We limit our interest in communication and use random synthetic data (of dimension (224, 224, 3)) as input to avoid impacts of I/O performance on the results. The training is in single precision with batch size of 64. We conducted the scalability experiments on the production code STEMDL using TensorFlow. We briefly discuss STEMDL in Section 2.3, report its source code in a GitHub repository (listed in Availability) and leave detailed documentation in Section 3 of the supplementary materials.

4.2 Evaluations on Horovod's Strategies

This section evaluates the performance of various strategies in Horovod. We compare the performance of caching and grouping to the existing strategies across scales. Figure 6 reports the results, in which we follow the definitions about the
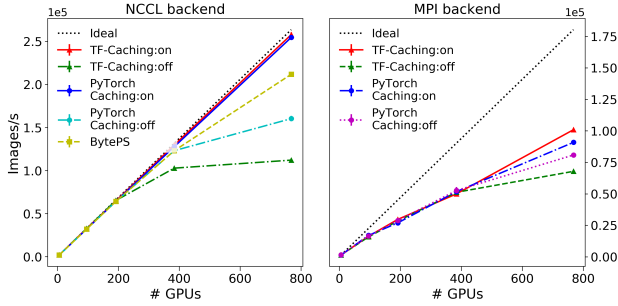
Figure 7: Performance of caching on ResNet50. We evaluate Horovod with caching enabled and disabled with both NCCL (left) and MPI (right) backends, and also compare the results to the performance of BytePS with NCCL (left).

existing strategies given in Section 2.3 and name the results of our techniques as `Caching` (cached-based coordination enabled) and `Caching+Grouping` (both caching and grouping enabled), respectively. Similar to Figure 3, we focus on analyzing performance (left subfigure) and GPU utilization (right subfigure). Here, performance refers to the the floating-point operations performed per second (FLOPs). It is clear that our solutions outperform the existing strategies across scales consistently. When increasing the number of GPUs in use, the performance advantage grows rapidly. In particular, at the scale of 6000 GPUs, `Caching+Grouping` and `Caching` obtain $1.97\times$ and $1.64\times$ GPU performance improvement, and equally $1.48\times$ utilization improvement, over the Horovod baseline in `NCCL-AllReduce`. Accelerated by our techniques, 175 petaFLOPS in FP16 precision (more detailed discussion can be seen in supplementary materials Section 2) can be delivered with less than a quarter of Summit.

We conclude that our techniques achieve better performance than the existing strategies, especially at scale.

4.3 Evaluations across Frameworks and Communication Libraries

Next, we evaluate caching and grouping with both Tensor-Flow and Pytorch, and compare our techniques to tf.distribute, torch.DDP, and BytePS.

4.3.1 Caching and Grouping across Frameworks

We first analyze the caching performance on Horovod with TensorFlow and Pytorch. Figure 7 presents the results. It suggests that, for the results with both NCCL and MPI, the caching-enabled Horovod (`TF-Caching:on` and `PyTorch Caching:on`) first delivers equally good performance; and when increasing the number of GPUs to 384 and more, the caching-enabled Horovod delivers better performance consistently with both TensorFlow and Pytorch. In particular, compared to the caching-disabled Horovod (`TF-Caching:off` and `PyTorch Caching:off`) with NCCL on 768 GPUs, the
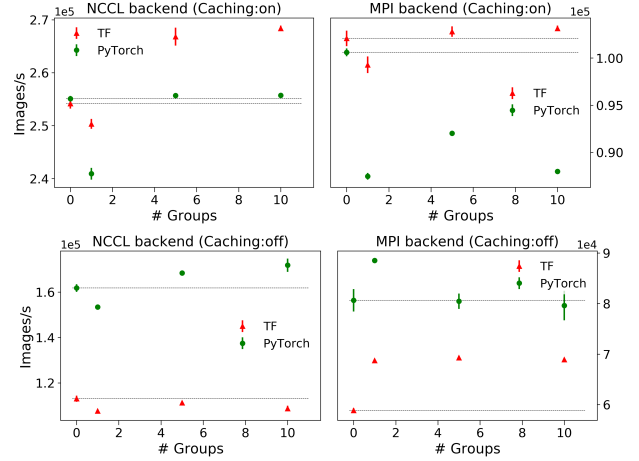

Figure 8: Performance of grouping on ResNet50. We evaluate Horovod with varied grouping configurations on 768 GPUs with caching enabled (top) and disabled (bottom) and with NCCL (left) and MPI (right) backends.

caching strategy achieves $2.5\times$ (`TF-Caching:on`) and $1.6\times$ (`PyTorch Caching:on`) performance improvement, respectively. Compared to the caching-disabled Horovod with MPI on 768 GPUs, the caching strategy achieves $1.53\times$ and $1.15\times$ performance improvement, respectively.

Figure 7 also presents the performance of BytePS (`BytePS`). It is shown clearly that BytePS delivers better performance than the cache-disabled Horovod consistently, and delivers equally good performance as the caching strategy does on the range of 6 GPUs — 384 GPUs, and delivers 20% lower performance than the caching strategy does on 768 GPUs. This suggests that, at larger scales, BytePS exhibits the scalability issue in typical HPC settings such as Summit. We leave the further study on the performance of BytePS on HPC clusters as future work.

Next, we report the grouping benefit in Figure 8. In the case with caching enabled (`Caching:on`) , comparing to the case without grouping (# groups = 0), the training throughput on 768 GPUs with Horovod (NCCL backend) obtains a decent 5% boost with 5 or 10 tensor groups for TensorFlow, although the gain for PyTorch is less significant. For the much slower MPI backend, the improvement becomes marginal or negative. When the caching is turned off (`Caching:off`) , there is a performance boost for PyTorch with the optimal group size, while for TensorFlow, it benefits mostly from grouping on the MPI backend. This indicates complicated interactions between the grouping and caching optimization.

To obtain a better understanding on the grouping behavior under different frameworks and communication fabrics, we plot the timing breakdown in Horovod for a 768-GPU training in Figure 9. For each iteration, the timing consists of two parts: coordination (control plane) and AllReduce (data plane). The timing for the AllReduce portion is further
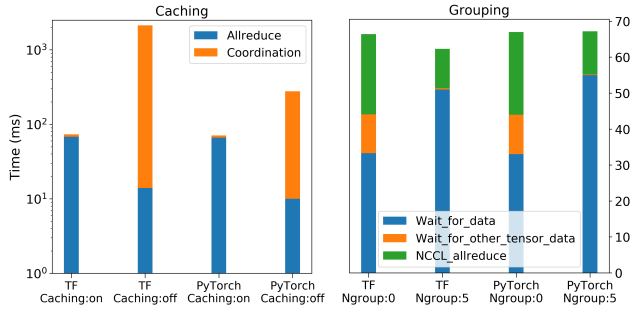
Figure 9: The inner timing breakdown in Horovod (NCCL backend) for a 768-GPU training with caching enabled and disabled (left) and grouping (# groups = 5) (right), respectively, during the training of ResNet50.
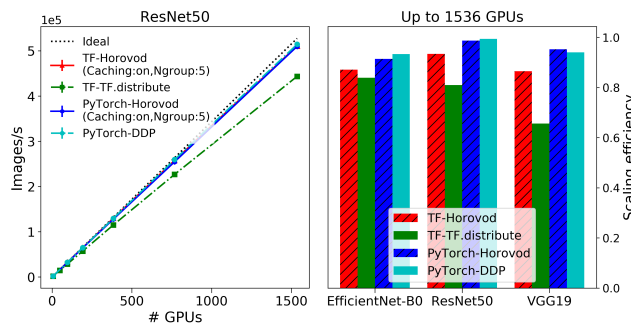

Figure 10: Scaling comparisons among Horovod, tf.distribute, and torch.DDP for the training of EfficientNet-B0, ResNet50, and VGG19. Training throughput (images/s) of ResNet50 (left). Scaling efficiency using up to 1536 GPUs (right).

split into wait (denoted [3] in Horovod as `WAIT_FOR_DATA` and `WAIT_FOR_OTHER_TENSOR_DATA` for time on waiting for framework to deliver gradient data and other data in the same fused collective, respectively) and actual communication (NCCL AllReduce). The case is slightly complicated for grouping. On one hand, the NCCL AllReduce time is almost cut in half because the grouped messages (orders of 10 MB) can better utilize network bandwidth; on the other hand, the wait time increases due to the coordination of groups. The overall performance of grouping depends on the trade-off between the aforementioned 2 factors. Too small number of groups (larger message and longer wait time) or too slow communication fabric (smaller or no gain in larger message communication) may result in worse performance with grouping, as indicated in Figure 8.

4.3.2 Evaluations across Communication Libraries

With both caching and grouping enabled, we compare the scaling efficiency of Horovod with tf.distribute and torch.DDP. To conduct a fair comparison, we ran all three libraries using a NCCL backend, and configure tf.distribute to use its

AllReduce mode (`MultiWorkerMirroredStrategy`), similar to Horovod and torch.DDP. In contrast to the experiments with TensorFlow v2.3.1 reported in the previous sections, this section contains experiments run using tf.distribute in TensorFlow v2.4 as it supports a comparable grouping feature and is a more recent release. Moreover, we disabled the `broadcast_buffers` option in torch.DDP to ensure that no additional collective operations outside the gradient AllReduces are performed during testing. We set the bucket size/pack size for grouping in torch.DDP and tf.distribute to 25MB as it is the default configuration for torch.DDP.

We present the results in Figure 10. As is shown clearly in the left subfigure, using up to 1536 GPUs, Horovod delivers 93% and 96% of scaling efficiencies with TensorFlow and PyTorch, respectively, while tf.distribute and DDP achieve 81% and 97% of the efficiencies, respectively. To further illustrate the scaling on different communication volumes, we plot the scaling efficiency for EfficientNet-B0, ResNet50, and VGG19 (right subfigure). Our approach shows an average of 12% better scaling than tf.distribute and a comparable performance to DDP, across model sizes, and the advantage becomes bigger as communication volume increases.

To obtain a better understanding of the performance of the three libraries, we profiled the training of ResNet50 with the libraries using Nsight Systems [4] (an NVIDIA profiling tool) and observed how well the AllReduce operations overlap with computation within a training iteration for each library. The results (see details in supplementary materials Section 4) show that all three libraries group tensors for AllReduce to a similar number of large buffers per iteration (4 or 5). In particular, we observed >95% of AllReduce overlapped with computation when using Horovod and torch.DDP, and the number dropped to ~75% when using tf.distribute.

We conclude that our solution performs well with both TensorFlow and PyTorch. Moreover, it delivers comparable and/or better performance than tf.distribute and torch.DDP, especially for large communication volumes.

4.4 Scaling Analysis on Production Code

This section evaluates the scaling efficiency of our solutions using a scientific DNN training code, STEMDL. The purpose of the section is to demonstrate a use case that stresses the communication layer of DL training at extreme scales (e.g. 27k GPUs). Our expectation is that if a communication implementation can scale well in this scenario, it should be well suited to many other workloads operating with far fewer tasks. Beyond scaling efficiency, we also evaluate the power consumption and overall performance of the production runs of STEMDL on the fully-scaled Summit, and leave the detailed documentation (e.g., the metrics and evaluations) to Section 2 in the supplementary materials, due to space limitations.

Figure 11 presents the scaling results. With both caching and grouping enabled, Horovod achieves a scaling efficiency
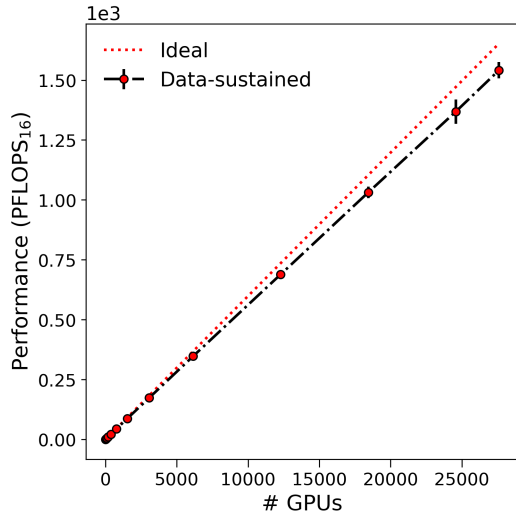
Figure 11: Scaling efficiency of STEMDL using up to 27,600 GPUs, the entire Summit.

of 0.93 at 27,600 GPUs and reach a sustained performance of 1.54 exaflops (with standard error +- 0.02) and a peak performance of 2.15 exaflops (with standard error +-0.02) in FP16 precision. Moreover, on a single GPU, our proposals attain 59.67 and 83.92 teraflops as the sustained and peak performance, respectively. It suggests that each GPU achieves 49.7% and 70% of the theoretical peak performance of a V100 (120 teraflops) as its sustained and peak performance. To the best of our knowledge, it exceeds the single GPU performance of all other DNN trained on the same system to date.

We conclude that our techniques can attain near-linear scaling on up to 27,600 GPUs.

5 Related Work

Other than collective AllReduce, another popular scheme for data parallelism is parameter server. Incorporated with many acceleration techniques such as hierarchical strategy, priority-based scheduling, etc, BytePS [12, 24] has shown better scaling performance than Horovod in a cloud environment where parameter servers run on CPU-only nodes, because the network bandwidth can be more efficiently utilized[4]. We compared our solutions with BytePS on a typical HPC setting and the results (see Figure 7) show that our techniques perform better in such settings.

One promising direction is to further reduce the communication volume via compression [8, 11, 26, 35, 36], decentralized learning [13, 14, 19], or staled/asynchronized communication [9, 10, 20]. The compression techniques include quantization, sparsification, sketching, etc, and the combined

---

[4]In current ring-based AllReduce (as implemented in NCCL), each model replica sends and receives $2(N-1)/N$ times gradients (N being number of GPUs), so the total message volume transferred in network per model is 2x of the gradient volume for large N.

method [22] has shown 2 orders of magnitude in communication volume reduction without loss of accuracy. For decentralized learning, depending on the communication graphs for model replicas, the communication complexity is reduced to $O(Deg(graph))$ independent of scale. Staled/asynchronized communication can boost the communication performance by relaxing the synchronization requirement across model replicas, which usually comes with some cost in model convergence. These developments are orthogonal to our approach, and in principle, our techniques can apply on top of them.

Beyond proposals for improving collective communication in DNN training. Kungfu [23] is proposed to auto-tune the parameters in both DNN models and DL frameworks based on runtime monitoring data. This effort is complementary to ours: we propose techniques in Horovod with introduction of parameters that may benefit tremendously from appropriate tuning. Another significant recent study [28] proposed Drizzle to improve large scale streaming systems with group scheduling and pre-scheduling shuffles. Similar to our approach, Drizzle reused scheduling decisions to reduce coordination overhead across micro-batches. But different to our decentralized coordination proposal, Drizzle amortized the overhead of centralized scheduling.

6 Conclusion

We have shown that by introducing a new coordination strategy and a grouping strategy we exceed the state of the art in scaling efficiency. This opens up, in particular, opportunities in exploiting the different levels of parallelism present in many systems (e.g. intra-node vs inter-node) such as Summit to train even larger DNN models.

Acknowledgments

Availability

The proposed techniques have been upstreamed to the Horovod main distribution [3]. The code for full Summit distributed training and the software for data generation are made public [16, 17].

References

[1] BytePS Best Practice. https://github.com/bytedance/byteps/blob/master/docs/best-practice.md.

[2] Gloo. https://github.com/facebookincubator/gloo.

[3] Horovod. https://github.com/horovod/horovod.

[4] Nvidia Nsight. https://developer.nvidia.com/nsight-systems.

[5] PyTorch. https://pytorch.org/.

[6] tf.distribute in TensorFlow. https://www.tensorflow.org/api_docs/python/tf/distribute.

[7] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, 2016.

[8] Naman Agarwal, Ananda Theertha Suresh, Felix Yu, Sanjiv Kumar, and H. Brendan McMahan. cpSGD: Communication-efficient and differentially-private distributed SGD. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS'18)*, 2018.

[9] Suyog Gupta, Wei Zhang, and Fei Wang. Model accuracy and runtime tradeoff in distributed deep learning: A systematic study. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI'17)*, 2017.

[10] Qirong Ho, James Cipar, Henggang Cui, Jin Kyu Kim, Seunghak Lee, Phillip B. Gibbons, Garth A. Gibson, Gregory R. Ganger, and Eric P. Xing. More effective distributed ML via a stale synchronous parallel parameter server. In *Proceedings of the 26th International Conference on Neural Information Processing Systems (NIPS'13)*, 2013.

[11] Nikita Ivkin, Daniel Rothchild, Enayat Ullah, Vladimir braverman, Ion Stoica, and Raman Arora. Communication-efficient distributed SGD with sketching. In *Advances in Neural Information Processing Systems (NIPS'19)*, 2019.

[12] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, 2020.

[13] Anastasia Koloskova*, Tao Lin*, Sebastian U Stich, and Martin Jaggi. Decentralized deep learning with arbitrary communication compression. In *Proceedings of the International Conference on Learning Representations (ICLR'20)*, 2020.

[14] Anastasia Koloskova, Sebastian U Stich, and Martin Jaggi. Decentralized stochastic optimization and gossip algorithms with compressed communication. In *Proceedings of the 36th International Conference on Machine Learning (ICML'19)*, 2019.

[15] Thorsten Kurth, Sean Treichler, Joshua Romero, Mayur Mudigonda, Nathan Luehr, Everett Phillips, Ankur Mahesh, Michael Matheson, Jack Deslippe, Massimiliano Fatica, Prabhat, and Michael Houston. Exascale deep learning for climate analytics. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'18)*, 2018.

[16] Nouamane Laanait, Michael A Matheson, Suhas Somnath, Junqi Yin, and USDOE. STEMDL. https://www.osti.gov//servlets/purl/1630730, 9 2019.

[17] Nouamane Laanait, Junqi Yin, and USDOE. NAMSA. https://www.osti.gov//servlets/purl/1631694, 8 2019.

[18] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. PyTorch distributed: Experiences on accelerating data parallel training. *Very Large Data Bases Conference (VLDB'20)*, 2020.

[19] Youjie Li, Mingchao Yu, Songze Li, Salman Avestimehr, Nam Sung Kim, and Alexander Schwing. Pipe-SGD: A decentralized pipelined SGD framework for distributed deep net training. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS'18)*, 2018.

[20] Xiangru Lian, Yijun Huang, Yuncheng Li, and Ji Liu. Asynchronous parallel stochastic gradient for nonconvex optimization. In *Proceedings of the 28th International Conference on Neural Information Processing Systems (NIPS'15)*, 2015.

[21] Xiangru Lian, Wei Zhang, Ce Zhang, and Ji Liu. Asynchronous decentralized parallel stochastic gradient descent. In *Proceedings of the 35th International Conference on Machine Learning (ICML'18)*, 2018.

[22] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and Bill Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. In *Proceedings of the International Conference on Learning Representations (ICLR'18)*, 2018.

[23] Luo Mai, Guo Li, Marcel Wagenländer, Konstantinos Fertakis, Andrei-Octavian Brabete, and Peter Pietzuch. Kungfu: Making training in distributed machine learning adaptive. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, 2020.

[24] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed DNN training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)*, 2019.

[25] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *CoRR*, abs/1802.05799, 2018.

[26] Ryan Spring, Anastasios Kyrillidis, Vijai Mohan, and Anshumali Shrivastava. Compressing gradient optimizers via count-sketches. In *Proceedings of the 36th International Conference on Machine Learning (ICML'19)*, 2019.

[27] Sudharshan S. Vazhkudai, Bronis R. de Supinski, Arthur S. Bland, Al Geist, James Sexton, Jim Kahle, Christopher J. Zimmer, Scott Atchley, Sarp Oral, Don E. Maxwell, Veronica G. Vergara Larrea, Adam Bertsch, Robin Goldstone, Wayne Joubert, Chris Chambreau, David Appelhans, Robert Blackmore, Ben Casses, George Chochia, Gene Davison, Matthew A. Ezell, Tom Gooding, Elsa Gonsiorowski, Leopold Grinberg, Bill Hanson, Bill Hartner, Ian Karlin, Matthew L. Leininger, Dustin Leverman, Chris Marroquin, Adam Moody, Martin Ohmacht, Ramesh Pankajakshan, Fernando Pizzano, James H. Rogers, Bryan Rosenburg, Drew Schmidt, Mallikarjun Shankar, Feiyi Wang, Py Watson, Bob Walkup, Lance D. Weems, and Junqi Yin. The design, deployment, and evaluation of the coral pre-exascale systems. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'18)*, 2018.

[28] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J Franklin, Benjamin Recht, and Ion Stoica. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*, 2017.

[29] Bing Xie, Jeffrey Chase, David Dillow, Oleg Drokin, Scott Klasky, Sarp Oral, and Norbert Podhorszki. Characterizing output bottlenecks in a supercomputer. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'12)*, 2012.

[30] Bing Xie, Jeffrey Chase, David Dillow, Scott Klasky, Jay Lofstead, Sarp Oral, and Norbert Podhorszki. Output performance study on a production petascale filesystem. In *HPC I/O in the Data Center Workshop (HPC-IODC'17)*, 2017.

[31] Bing Xie, Yezhou Huang, Jeffrey Chase, Jong Youl Choi, Scott Klasky, Jay Lofstead, and Sarp Oral. Predicting output performance of a petascale supercomputer. In *Proceedings of the International ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC'17)*, 2017.

[32] Bing Xie, Sarp Oral, Christopher Zimmer, Jong Youl Choi, David Dillow, Scott Klasky, Jay Lofstead, Norbert Podhorszki, and Jeffrey S Chase. Characterizing output bottlenecks of a production supercomputer: Analysis and implications. *ACM Transactions on Storage (TOS'20)*, 2020.

[33] Bing Xie, Zilong Tan, Phil Carns, Jeff Chase, Kevin Harms, Jay Lofstead, Sarp Oral, Sudharshan Vazhkudai, and Feiyi Wang. Applying machine learning to understand write performance of large-scale parallel filesystems. In *the 4TH International Parallel Data Systems Workshop (PDSW'19)*, 2019.

[34] Bing Xie, Zilong Tan, Phil Carns, Jeff Chase, Kevin Harms, Jay Lofstead, Sarp Oral, Sudharshan S Vazhkudai, and Feiyi Wang. Interpreting write performance of supercomputer I/O systems with regression models. In *Proceedings of the 36th IEEE International Parallel and Distributed Processing Symposium (IPDPS'21)*, 2021.

[35] Min Ye and Emmanuel Abbe. Communication-computation efficient gradient coding. In *Proceedings of the 35th International Conference on Machine Learning (ICML'18)*, 2018.

[36] Yue Yu, Jiaxiang Wu, and Longbo Huang. Double quantization for communication-efficient distributed optimization. In *Advances in Neural Information Processing Systems (NIPS'19)*, 2019.

Cocktail: A Multidimensional Optimization for Model Serving in Cloud

Jashwant Raj Gunasekaran, Cyan Subhra Mishra, Prashanth Thinakaran, Bikash Sharma,
Mahmut Taylan Kandemir, Chita R. Das

The Pennsylvania State University, University Park, PA

{jashwant, cyan, prashanth, mtk2, cxd12}@psu.edu, bikash.nitrkl@acm.org

Abstract

With a growing demand for adopting ML models for a variety of application services, it is vital that the frameworks serving these models are capable of delivering highly accurate predictions with minimal latency along with reduced deployment costs in a public cloud environment. Despite high latency, prior works in this domain are crucially limited by the accuracy offered by individual models. Intuitively, model ensembling can address the accuracy gap by intelligently combining different models in parallel. However, selecting the appropriate models dynamically at runtime to meet the desired accuracy with low latency at minimal deployment cost is a nontrivial problem. Towards this, we propose *Cocktail*, a cost effective ensembling-based model serving framework. *Cocktail* comprises of two key components: (i) a dynamic model selection framework, which reduces the number of models in the ensemble, while satisfying the accuracy and latency requirements; (ii) an adaptive resource management (RM) framework that employs a distributed proactive autoscaling policy, to efficiently allocate resources for the models. The RM framework leverages transient virtual machine (VM) instances to reduce the deployment cost in a public cloud. A prototype implementation of *Cocktail* on the AWS EC2 platform and exhaustive evaluations using a variety of workloads demonstrate that *Cocktail* can reduce deployment cost by $1.45\times$, while providing $2\times$ reduction in latency and satisfying the target accuracy for up to 96% of the requests, when compared to state-of-the-art model-serving frameworks.

1 Introduction

Machine Learning (ML) has revolutionized user experience in various cloud-based application domains such as product recommendations [70], personalized advertisements [44], and computer vision [13, 43]. For instance, Facebook [44, 82] serves trillions of inference requests for user-interactive applications like ranking new-feeds, classifying photos, etc. It is imperative for these applications to deliver accurate predictions at sub-millisecond latencies [27,34,35,39,44,83] as they critically impact the user experience. This trend is expected to perpetuate as a number of applications adopt a variety of ML models to augment their services. These ML models are typically trained and hosted on cloud platforms as service endpoints, also known as *model-serving* framework [6, 28, 60]. From the myriad of ML flavours, Deep Neural Networks

(DNNs) [54] due to their multi-faceted nature, and highly generalized and accurate learning patterns [45,73] are dominating the landscape by making these model-serving frameworks accessible to developers. However, their high variance due to the fluctuations in training data along with compute and memory intensiveness [59,65,84] has been a major impediment in designing models with high accuracy and low latency. Prior model-serving frameworks like InFaas [83] are confined by the accuracy and latency offered by such individual models.

Unlike single-model inferences, more sophisticated techniques like *ensemble learning* [15] have been instrumental in allowing model-serving to further improve accuracy with multiple models. For example, by using the ensembling [1] technique, images can be classified using multiple models *in parallel* and results can be combined to give a final prediction. This significantly boosts accuracy compared to single-models, and for this obvious advantage, frameworks like Clipper [27] leverage ensembling techniques. Nevertheless, with ensembling, the very high resource footprint due to sheer number of models that need to be run for each request [27, 56], exacerbates the public cloud deployment costs, as well as leads to high variation in latencies. Since cost plays a crucial role in application-provider consideration, it is quintessential to minimize the deployment costs, while maximizing accuracy with low latency. Hence, the non-trivial challenge here lies in making the cost of ensembling predictions analogous to single model predictions, while satisfying these requirements.

Studying the state-of-the-art ensemble model-serving frameworks, we observe the following critical shortcomings:

• Ensemble model selection policies used in frameworks like Clipper [27] are static, as they *ensemble all available models* and focus solely on minimizing loss in accuracy. This leads to higher latencies and further inflates the resource footprint, thereby accentuating the deployment costs.

• Existing ensemble weight estimation [87] has *high computational complexity* and in practice is limited to a small set of off-the-shelf models. This leads to significant loss in accuracy. Besides, employing linear ensembling techniques such as model averaging are compute intensive [80] and not scalable for a large number of available models.

• Ensemble systems [27,80] are *not focused towards model deployment* in a public cloud infrastructure, where resource

---

[1]We refer to ensemble-learning as ensembling throughout the paper.

selection and procurement play a pivotal role in minimizing the latency and deployment costs. Further, the resource provisioning strategies employed in single model-serving systems are *not directly extendable* to ensemble systems.

These shortcomings collectively motivate the central premise of this work: *how to solve the complex optimization problem of cost, accuracy and latency for an ensembling framework?* In this paper, we present and evaluate *Cocktail*[2], which to our knowledge is the first work that proposes a cost-effective model-serving system by exploiting ensembling techniques for classification-based inference, to deliver high accuracy and low latency predictions. *Cocktail* adopts a three-pronged approach to solve the optimization problem. First, it uses a dynamic model selection policy to significantly reduce the number of models used in an ensemble, while meeting the latency and accuracy requirements.
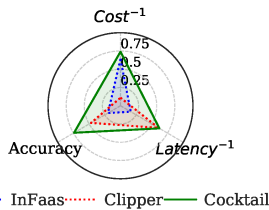

Figure 1: Benefits of *Cocktail*. Results are normalized (higher the better).

Second, it utilizes distributed autoscaling policies to reduce the latency variability and resource consumption of hosting ensemble models. Third, it minimizes the cost of deploying ensembles in a public cloud by taking advantage of transient VMs, as they can be 70-90% cheaper [3] than traditional VMs. *Cocktail*, by coalescing these benefits, is capable of operating in a region of optimal cost, accuracy and latency (shown in Figure 1) that prior works cannot achieve. Towards this, the **key contributions** of the paper are summarized below:

1. By characterizing accuracy *vs.* latency of ensemble models, we identify that prudently selecting a subset of available models under a given latency can achieve the target accuracy. We leverage this in *Cocktail*, to design a novel dynamic model selection policy, which ensures accuracy with significantly reduced number of models.

2. Focusing on classification-based inferences, it is important to minimize the bias in predictions resulting from multiple models. In *Cocktail*, we employ a per-class weighted majority voting policy, that makes it scalable and effectively breaks ties when compared to traditional weighted averaging, thereby minimizing the accuracy loss.

3. We show that uniformly scaling resources for all models in the ensemble leads to over-provisioning of resources and towards minimizing it, we build a distributed weighted auto-scaling policy that utilizes the *importance sampling* technique to proactively allocate resources to every model. Further, *Cocktail* leverages transient VMs as they are cheaper, to drastically minimize the cost for hosting model-serving infrastructure in a public cloud.

---

[2]*Cocktail* is ascribed to having the perfect blend of models in an ensemble.
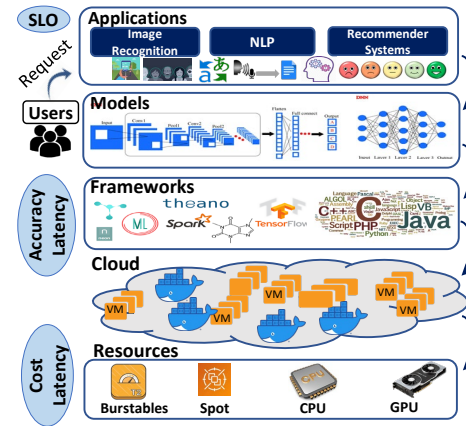

Figure 2: The overall framework for model-serving in public cloud.

4. We implement a prototype of *Cocktail* using both CPU and GPU instances on AWS EC2 [5] platform and extensively evaluate it using different request-arrival traces. Our results from exhaustive experimental analysis demonstrate that *Cocktail* can minimize deployment cost by 1.4× while meeting the accuracy for up-to 96% of the requests and providing 2× reduction in latency, when compared to state-of-the-art model serving systems.

5. We show that ensemble models are inherently fault-tolerant over single models, since in the former, failure of a model would incur some accuracy loss without complete failure of the requests. It is observed from our failure-resilience results that *Cocktail* can adapt to instance failures by limiting the accuracy loss within 0.6%.

2 Background and Motivation

We start by providing a brief overview of model-serving in public cloud and ensembling, followed by a detailed analysis of their performance to motivate the need for *Cocktail*.

2.1 Model Serving in Public Cloud

Figure 2 shows the overall architecture of a model-serving framework. There are diverse applications that are typically developed, trained and hosted as web services. These services allow end-users to submit queries via web server interface. Since these inference requests are often user-facing, it is imperative to administer them under a strict service level objective (SLO). We define SLO as the end-to-end response latency required by an application. Services like Ads and News Feed [39, 44] would require SLOs within 100ms, while facial tag recommendation [83] can tolerate up to 1000ms. A myriad of model architectures are available to train these applications which by themselves can be deployed on application frameworks like `TensorFlow` [1], `PyTorch` [62] etc. Table 1 shows the different models available for image prediction, that are pretrained on Keras using `ImageNet` [29] dataset. Each model has unique accuracy and latencies depending on the model architecture. Typically denser models are designed with more parameters (ex. `NASLarge`) to classify complex

Model (Acronym)	Params (10k)	Top-1 Accuracy(%)	Latency (ms)	P_f
MobileNetV1 (MNet)	4,253	70.40	43.45	10
MobileNetV2 (MNetV2)	4,253	71.30	41.5	10
NASNetMobile (NASMob)	5,326	74.40	78.18	3
DenseNet121 (DNet121)	8,062	75.00	102.35	3
DenseNet201 (DNet201)	20,242	77.30	152.21	2
Xception (Xcep)	22,910	79.00	119.2	4
Inception V3 (Incep)	23,851	77.90	89	5
ResNet50-V2 (RNet50)	25,613	76.00	89.5	6
Resnet50 (RNet50)	25,636	74.90	98.22	5
IncepResnetV2 (IRV2)	55,873	80.30	151.96	1
NasNetLarge (NasLarge)	343,000	82.00	311	1

Table 1: Collection of pretrained models used for image classification.

Features	Clipper [27]	Rafiki [80]	Infaas [83]	MArk [86]	Sagemaker	Swayam [34]	Cocktail
Predictive Scaling	✗	✗	✗	✓	✗	✓	✓
SLO Guarantees	✓	✗	✓	✓	✗	✓	✓
Cost Effective	✗	✗	✓	✓	✗	✗	✓
Ensembling	✓	✓	✗	✗	✓	✗	✓
Heterogeneous Instances	✗	✓	✓	✓	✓	✗	✓
Dynamic ensemble selection	✗	✗	✗	✗	✗	✗	✓
Model abstraction	✓	✓	✓	✗	✗	✗	✓

Table 2: Comparing *Cocktail* with other related frameworks.

classes of images. These 11 models are a representative set to classify all images belonging to 1000 classes in `Imagenet`. Depending on the application type, the maximum ensemble size can vary from tens to hundreds of models.

The entire model framework is typically hosted on resources like VMs or containers in public cloud. These resources are available in different types including CPU/GPU instances, burstables and transient instances. Transient instances [69] are similar to traditional VMs but can be revoked at any time by the cloud provider with an interruption notice. The provisioning latency, instance permanence and packing factor of these resources have a direct impact on the latency and cost of hosting model-serving. We explain instance "packing factor" and its relationship with latency in Section 2.3.2. In this paper, we focus on improving the accuracy and latency from the model selection perspective and consider instances types from a cost perspective. A majority of the model serving systems [6, 83, 86] in public cloud support individual model selection from available models. For instance, InFaas [83] can choose variants among a same model to maintain accuracy and latency requirements. However, denser models tend to have up to 6× the size and twice the latency of smaller models to achieve increased accuracy of about 2-3%. Besides using dense models, ensembling [15] techniques have been used to achieve higher accuracy.

Why Ensembling? An Ensemble is defined as a set of classifiers whose individual decisions combined in some way to classify new examples. This has proved to be more accurate than traditional single large models because it inherently reduces incorrect predictions due to variance and bias. The commonly used ensemble method in classification problems is bagging [33] that considers homogeneous weak learners, learns them independently from each other in parallel, and combines them following some kind of deterministic averaging process [18] or majority voting [49] process. For further details on ensemble models, we refer the reader to prior works [14, 57, 58, 61, 64, 77, 78, 88].

2.2 Related Work

Ensembling in practice: Ensembling is supported by commercial cloud providers like Azure ML-studio [11] and AWS Autogluon [31] to boost the accuracy compared to single models. Azure initially starts with 5 models and scales up to 200 using a hill-climb policy [17] to meet the target accuracy. AWS combines about 6-12 models to give the best possible accuracy. Users also have the option to manually mention the ensemble size. Unlike them, *Cocktail's* model selection policy tries to right-size the ensemble for a given latency, while maximizing accuracy.

Model-serving in Cloud: The most relevant prior works to *Cocktail* are InFaas [83] and Clipper [27], which have been extensively discussed and compared to in Section 6. Recently FrugalML [20] was proposed to cost-effectively choose from commercial MLaaS APIs. While striking a few similarities with *Cocktail*, it is practically limited to image-classification applications with very few classes and does not address resource provisioning challenges. Several works [37, 38] like MArk [86] proposed SLO and cost aware resource procurement policies for model-serving. Although our heterogeneous instance procurement policy has some similarities with MArk, it is significantly different because we consider ensemble models. Rafiki [80] considers small model sets and scales up and down the ensemble size by trading off accuracy to match throughput demands. However, *Cocktail's* resource management is more adaptive to changing request loads and does not drop accuracy. Pretzel [52] and Inferline [26] are built on top of Clipper to optimize the prediction pipeline and cost due to load variations, respectively. Many prior works [2, 25, 35, 63, 74, 75] have extensively tried to reduce model latency by reducing overheads due to shared resources and hardware interference. We believe that our proposed policies can be complementary and beneficial to these prior works to reduce the cost and resource footprint of ensembling. There are mainstream commercial systems which automate single model-serving like TF-Serving [60], SageMaker [6], AzureML [10], Deep-Studio [28] etc.

Autoscaling in Public Cloud: There are several research works that optimize the resource provisioning cost in public cloud. These works are broadly categorized into: (i) multiplexing the different instance types (e.g., Spot, On-Demand) [12, 23, 34, 41, 42, 68, 79], (ii) proactive resource provisioning based on prediction policies [34,36,40,41,69,86]. *Cocktail* uses similar load prediction models and auto-scales VMs in a distributed fashion with respect to model ensembling. Swayam [34] is relatively similar to our work as it han-

Baseline(BL)	NASLarge	IRV2	Xception	DNet121	NASMob
#Models	10	8	7	5	2
BL_Latency	311(ms)	152(ms)	120(ms)	100(ms)	98(ms)
E_Latency	152(ms)	120(ms)	103(ms)	89(ms)	44(ms)

Table 3: Comparing latency of Ensembling (E_Latency) with single (baseline) models.

dles container provisioning and load-balancing, specifically catered for single model inferences. *Cocktail's* autoscaling policy strikes parallels with Swayam's distributed autoscaling; however, we further incorporate novel importance sampling techniques to reduce over-provisioning for under-used models. Table 2 provides a comprehensive comparison of *Cocktail* with the most relevant works across key dimensions.

2.3 Pros and Cons of Model Ensembling

In this section, we quantitatively evaluate (i) how effective ensembles are in terms of accuracy and latency compared to single models, and (ii) the challenges in deploying ensemble frameworks in a cost-effective fashion on a public cloud. For relevance in comparison to prior work [27, 83] we chose image inference as our ensemble workload. While ensembling is applicable in other classification workloads like product recommendations [24, 53], text classification [71] etc, the observations drawn are generic and applicable to other applications.

2.3.1 Ensembling Compared to Single Models

To analyze the accuracy offered by ensemble models, we conduct an experiment using 10000 images from ImageNet [29] test dataset, on a C5.xlarge [8] instances in AWS EC2 [5]. For a given baseline model, we combine all models whose latency is lower than that of the baseline, and call it full-ensemble. We perform ensembling on the predictions using a simple majority voting policy. The latency numbers for the baseline models and the corresponding ensemble models along with the size of the ensemble are shown in Table 3. In majority voting, every model votes for a prediction for each input, and the final output prediction is the one that receives more than half of the votes. Figure 3a, shows the accuracy comparison of the baseline (single) and static ensemble (explained in Section 3) compared to the full-ensemble. It is evident that full-ensemble can achieve up to 1.65% better accuracy than single models.

Besides accuracy again, ensembling can also achieve lower latency. The latency of the ensemble is calculated as the time between start and end of the longest running model.As shown in Table 3, in the case of NASLarge, the ensemble latency is 2× lower (151ms) than the baseline latency (311ms). Even a 10ms reduction in latency is of significant importance to the providers [35]. We observe a similar trend of higher ensemble accuracy for other four baseline models with a latency reduction of up to 1.3×. Thus, depending on the model subset used in the ensemble, it achieves better accuracy than the baseline at lower latencies. Note that in our example model-set, the benefits of ensembling will diminish for lower
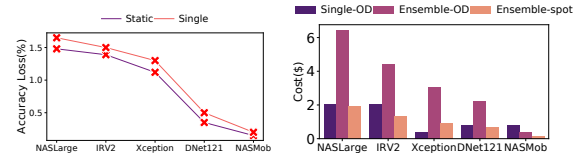
(a) Accuracy loss compared to full-ensemble. **(b)** Cost of full-ensembling hosted on OD and Spot instances.

Figure 3: Cost and accuracy of ensembling vs single models.

accuracies (< 75%) because single models can reach those accuracies. Hence, based on the user constraints, *Cocktail* chooses between ensemble and single models.

2.3.2 Ensembling Overhead

While ensembling can boost accuracy with low latency, their distinctive resource hungry nature drastically increases the deployment costs when compared to single models. This is because more VMs or containers have to be procured to match the resource demands. However, note that the "Packing factor" (P_f) for each model also impacts the deployment costs. P_f in this context is defined as the number of inferences that can be executed concurrently in a single instance without violating the inference latency (on average). Table 1 provides the P_f for 11 different models when executed on a C5.xlarge instance. There is a linear relationship between P_f and the instance size. It can be seen that smaller models (MNet, NASMob) can be packed 2-5× more when compared to larger models (IRV2, NASLarge). Thus, the ensembles with models of higher P_f have significantly lower cost.

The benefits of P_f is contingent upon the models chosen by the model selection policy. Existing ensemble model selection policies used in systems like Clipper use all off-the-shelf models and assign weights to them to calculate accuracy. However, they do not right-size the model selection to include models which primarily contribute to the majority voting. We compare the cost of hosting ensembles using both spot (ensemble-spot) and OD (ensemble-OD) instances with the single models hosted on OD (single-OD) instances. Ensemble-spot is explained further in the next section. We run the experiment over a period of 1 hour for 10 requests/second. The cost is calculated as the cost per hour of EC2 c5.xlarge instance use, billed by AWS [5]. We ensure all instances are fully utilized by packing multiple requests in accordance to the P_f. As shown in Figure 3b, Ensemble-OD is always expensive than single-OD for the all the models. Therefore, it is important to ensemble an "optimal" number of less compute intensive models to reduce the cost.

3 Prelude to Cocktail

To specifically address the cost of hosting an ensembling-based model-serving framework in public clouds without sacrificing the accuracy, this section introduces an overview of the two primary design choices employed in *Cocktail*.

How to reduce resource footprint? The first step towards making model ensembling cost effective is to minimize the

number of models by pruning the ensemble, which reduces the overall resource footprint. In order to estimate the right number of models to participate in a given ensemble, we conduct an experiment where we chose top $\frac{N}{2}$ accurate models (static) from the full-ensemble of size N. From Figure 3a, it can be seen that the static policy has an accuracy loss of up to 1.45% when compared to full-ensemble, but is still better than single models. This implies that the models other than top $\frac{N}{2}$ yields a significant 1.45% accuracy improvement in the full-ensemble but they cannot be statically determined.
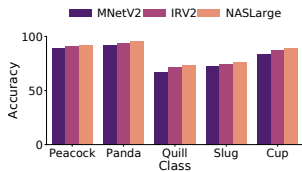
Figure 4: Class-wise Accuracy.

Therefore, a full-ensemble model participation is not required for all the inputs because, every model is individually suited to classify certain classes of images when compared to other classes. Figure 4 shows the class-wise accuracy for three models on 5 distinct classes. It can be seen that for simpler classes like Slug, `MNetV2` can achieve similar accuracy as the bigger models, while for difficult classes, like Cup and Quill, it experiences up to 3% loss in accuracy. Since the model participation for ensembling can vary based on the class of input images being classified, there is a scope to develop a dynamic model selection policy that can leverage this class-wise variability to intelligently determine the number of models required for a given input.

Key Takeaway: *Full ensemble model-selection is an overkill, while static-ensemble leads to accuracy loss. This calls for a dynamic model selection policy which can accurately determine the number of models required, contingent upon the accuracy and scalability of the model selection policy.*

How to save cost? Although dynamic model selection policies can significantly reduce the resource footprint as shown in Figure 3b, the cost is still 20-30% higher when compared to a single model inference. Most cloud providers offer transient VMs such as Amazon Spot instances [69], Google Preemptible VMs [9], and Azure Low-priority VMs [7], that can reduce cloud computing costs by as much as $10\times$ [3]. In *Cocktail*, we leverage these transient VMs such as spot instances to drastically reduce the cost of deploying ensembling model framework. As an example, we host full-ensembling on AWS spot instances. Figure 3b shows that ensemble-spot can reduce the cost by up to $3.3\times$ when compared to ensemble-OD. For certain baselines like IRV2, ensemble-spot is also $1.5\times$ cheaper than single-OD. However, the crucial downside of using transient VMs is that they can be unilaterally preempted by the cloud provider at any given point due to reasons like increase in bid-price or provider-induced random interruptions. As we will discuss further, *Cocktail* is resilient to instance failures owing to the fault-tolerance of ensembling by computing multiple inferences for a single request.

Key takeaway: *The cost-effectiveness of transient instances, is naturally suitable for hosting ensemble models.*
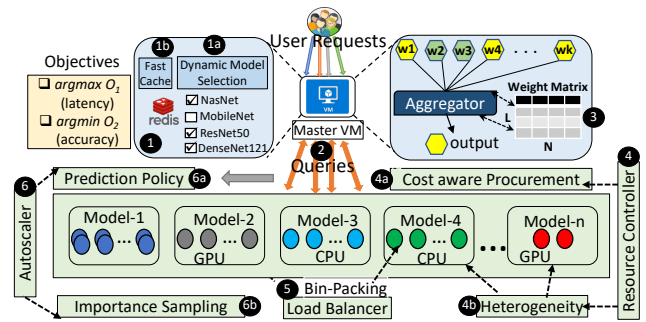

Figure 5: High-level overview of *Cocktail* design.

4 Overall Design of Cocktail

Motivated by our observations, we design a novel model-serving framework, *Cocktail*, that can deliver high-accuracy and low-latency predictions at reduced cost. Figure 5 depicts the high-level design of *Cocktail*. Users submit requests to a master VM, which runs a model selection algorithm, ❶a to decide the models to participate in the ensemble. The participating models are made available in a model cache ❶b for faster access and avoid re-computation for requests having similar constraints. Then, individual queries are dispatched to instances pools ❷ dedicated for each model. The results from the workers are *ensembled* using an weighted majority voting aggregator ❸ to agree upon a correct prediction. To efficiently address the resource management and scalability challenges, *Cocktail* applies multiple strategies.

First, it maintains dedicated instance pools to serve individual models which simplifies the management and load balancing overheads for every model. Next, the resource controller ❹ handles instance procurement, by exploiting both CPU and GPU instances ❹a in a cost-aware ❹b fashion, while the load balancer ❺ ensures all procured instances are bin-packed by assigning queries to appropriate instances. We also design an autoscaler ❻, which utilizes a prediction policy ❻a to forecast the request load and scale instances for every model pool, thereby minimizing over-provisioning of resources. The autoscaler further employs an importance sampling ❻b algorithm to estimate the importance of each model pool by calculating percentage of request served by it in a given time interval. The key components of the design are explained in detail below.

4.1 Dynamic Model Selection Policy

We use a window-based dynamic model selection policy using two objective functions as described below.

Objective functions: In order to reduce cost and latency while maximizing the accuracy, we define a latency-accuracy metric (μ_{AL}) and cost metric (μ_c):

$$\mu_{AL} = \frac{Acc_{target}}{Lat_{target}} \qquad \mu_C = k \times \sum_{m=1}^{N} \frac{inst\_cost}{P_{f_m}}$$

where N is the number of models used to ensemble and *inst_cost* is the VM cost. Each model m has a packing factor

P_{f_m} and k is a constant which depends on the VM size in terms of vCPUs (xlarge, 2xlarge, etc). Our first objective function (O_1) is to the maximize μ_{AL} such that target accuracy (Acc_{target}) is reached within the target latency (Lat_{target}).

$$\max \mu_{AL} : \begin{cases} Acc_{target} \geq Acc_{target} \pm Acc_{margin} \\ Lat_{target} \leq Lat_{target} \pm Lat_{margin} \end{cases}$$

To solve O_1, we determine an initial model list by choosing the individual models satisfying Lat_{target} and then create a probabilistic ensemble that satisfies the Acc_{target}. *Cocktail* takes the accuracy of each model as a probability of correctness and then iteratively constructs a model list, where the joint probability of them performing the classification is within the accuracy target. We tolerate a 0.2% (Acc_{margin}) and 5ms (Lat_{margin}) variance in Acc_{target} and Lat_{target}, respectively. Next, we solve for the second objective function (O_2) by minimizing μ_C, while maintaining the target accuracy.

$$\min \mu_C : \begin{cases} Acc_{target} \geq Acc_{target} \pm Acc_{margin} \end{cases}$$

(O_2) is solved by resizing the model list of size N and further through intelligence resource procurement (described in section 4.2), and thus maximizing P_f and minimizing k simultaneously. For N models, where each model has a minimum accuracy 'a', we model the ensemble as a coin-toss problem, where N biased coins (with probability of head being a) are tossed together, and we need to find the probability of majority of them being heads. For this, we need at least $\lfloor \frac{N}{2} \rfloor + 1$ models to give the same results. The probability of correct prediction is given by

$$\sum_{i=\lfloor \frac{N}{2} \rfloor + 1}^{N} \binom{N}{i} a^i (1-a)^{(N-i)}$$

Model Selection Algorithm: To minimize μ_C, we design a policy to downscale the number of models, if more than N/2+1 models vote for the same classification result. Algorithm 1 describes the overall design of the model selection policy ❶ₐ. For every monitoring interval, we keep track of the accuracy obtained from predicting all input images within the interval. If the accuracy of the interval reaches the threshold accuracy (target + error_margin), we scale down the number of available models in the ensemble. For consecutive sampling intervals, we calculate the Mode (most frequently occurring) of the majority vote received for every input. If the Mode is greater than needed votes $\lfloor N/2 \rfloor + 1$ we prune the models to $\lfloor N/2 \rfloor + 1$. While down-scaling, we drop the models with the least prediction accuracy in that interval. If there is a tie, we drop the model with least packing factor (P_f). It can so happen that dropping models can lead to drop in accuracy for certain intervals, because the class of images being predicted are different. In such cases, we up-size the models (one at a time) by adding most accurate model from the remaining unused models.

Algorithm 1 Model Selection and Weighted Majority Voting

1: **procedure** FULL_ENSEMBLE(MODELLIST, SLO)
2:     **for** model ∈ ModelList **do**
3:         **if** model.latency ≤ SLO.latency **then**
4:             Model.add(model)
5:         **end if**
6:     **end for** ⟨O_1⟩
7: **end procedure**
8: **procedure** DYNAMIC_MODEL_SCALING(*Models*)
9:     **if** *curr_accuracy* ≥ *accuracy_threshold* **then**
10:         **if** $max_{vote} > \frac{N}{2} + 1$ **then** ⟨O_2⟩
11:             $to\_be\_dropped \leftarrow max_{vote} - \frac{N}{2} + 1$
12:             $Models.drop(to\_be\_dropped)$
13:         **end if**
14:     **else**
15:         $addModel \leftarrow find\_models(remaining\_models)$
16:         $Models.append(addModel)$
17:     **end if**
18: **end procedure**
19: **procedure** WEIGHTED_VOTING(*Models*)
20:     **for** *model in ∀Models* **do**
21:         $class \leftarrow model.predicted\_class$
22:         $weighted\_vote[class] += weights[model.class]$
23:     **end for**
24:     $P_{class} \leftarrow \max(weighted\_vote, key = class)$
25:     $return P_{class}$
26: **end procedure**

4.1.1 Class-based Weighted Majority Voting

The model selection policy described above ensures that we only use the necessary models in the majority voting. In order to increase the accuracy of majority voting, we design a weighted majority voting policy ❸. The weight matrix is designed by considering the accuracy of each model for each class, giving us a weight matrix of $L \times N$ dimension, where L is the number of unique labels and N is the number of models used in the ensemble. The majority vote is calculated as a sum of model-weights for each unique class in the individual prediction of the ensemble. For instance, if there are 3 unique classes predicted by all the ensemble models, we sum the weights for all models of the same class. The class with the maximum weight (P_{class}) is the output of the majority vote. Hence, classes that did not get the highest votes can still be the final output if the models associated with that class has a higher weight, than the combined weights of highest voted class. Unlike commonly used voting policies which assign weights based on overall correct predictions, our policy incorporates class-wise information to the weights, thus making it more adaptable to different images classes.

In order to determine the weight of every class, we use a per-class dictionary that keeps track of the correct predictions of every model per class. We populate the dictionary at runtime to avoid any inherent bias that could result from varying images over time. Similarly, our model selection policy is also changed at runtime based on correct predictions seen during every interval. An important concern in majority voting is tie-breaking. Ties occur when two sets of equal number of models predict a different result. The effectiveness

---

Algorithm 2 Predictive Weighted Instance Auto Scaling

```
1:  procedure WEIGHTED_AUTOSCALING(Stages)
2:      Predicted_load ← DeepARN_Predict(load)
3:      for every Interval do
4:          for model in ∀Models do
5:              model_weight ← get_popularity(model)
6:              Weight.append(model_weight)
7:          end for
8:      end for
9:      if Predicted_load ≥ Current_load then
10:         for model in ∀Models do
11:             I_n ← (Predicted_load - Current_load) × model_weight
12:             launch_workers(est_VMs)
13:             model.workers.append(est_VMs)
14:         end for
15:     end if
16: end procedure
```

of weighted voting in breaking ties is discussed in Section 6.

4.2 Resource Management

Besides model selection, it is crucial to design an optimized resource provisioning and management scheme to host the models cost-effectively. We explain in detail the resource procurement and autoscaling policy employed in *Cocktail*.

4.2.1 Resource Controller

Resource controller determines the cost-effective combination of instances to be procured. We explain the details below.
Resource Types: We use both CPU and GPU instances ➍a depending on the request arrival load. GPU instances are cost-effective when packed with a large batch of requests for execution. Hence, inspired from prior work [27, 86], we design an adaptive packing policy such that it takes into account the number of requests to schedule at time T and P_f for every instance. The requests are sent to GPU instances only if the load matches the P_f of the instance.

Cost-aware Procurement: The cost of executing in a fully packed instance determines how expensive is each instance. Prior to scaling-up instances, we need to estimate the cost ➍b of running them along with existing instances. At any given time T, based on the predicted load (L_p) and running instances R_N, we use a cost-aware greedy policy to determine the number of additional instances required to serve as $A_n = L_p - C_r$, where $C_r = \sum_{i=1}^{N} P_{f_i}$, is the request load which can be handled with R_N. To procure A_n instances, we greedily calculate the least cost instance as $\min_{\forall i \in instances} Cost_i \times A_n / P_{f_i}$. Depending on the cost-effectiveness ratio of A_n/P_{f_i}, GPUs will be preferred over CPU instances.

Load Balancer: Apart from procuring instances, it is quintessential to design a load balancing and bin-packing ➎ strategy to fully utilize all the provisioned instances. We maintain a request queue at every model pool. In order to increase the utilization of all instances in a pool at any given time, the load balancer submits every request from the queue to the lease remaining free slots (viz. instance packing factor P_f). This is similar to an online bin-packing algorithm. We use an idle-timeout limit for 10 minutes to recycle unused

instances from every model pool. Hence, greedily assigning requests enables early scale down of lightly loaded instances.

4.2.2 Autoscaler

Along with resource procurement, we need to autoscale instances to satisfy the incoming query load. Though reactive policies (used in Clipper and InFaas) can be employed which take into account metrics like CPU utilization [83], these policies are slow to react when there is dynamism in request rates. Proactive policies with request prediction are know to have superior performance [86] and can co-exist with reactive policies. In *Cocktail*, we use a load prediction model that can accurately forecast the anticipated load for a given time interval. Using the predicted load ➏a, *Cocktail* spawns additional instances, if necessary, for every instance pool. In addition, we sample SLO violations for every 10s interval and reactively spawn additional instances to every pool based on aggregate resource utilization of all instances. This captures SLO violations due to mis-predictions.

Prediction Policy: To effectively capture the different load arrival patterns, we design a DeepAR-estimator (DeepARest) based prediction model. We zeroed in on the choice of using DeepARest by conducting (Table 4) an in-depth comparison of the accuracy loss when compared with other

Model	RMSE
MWA	77.5
EWMA	88.25
Linear R.	87.5
Logsitic R.	78.34
Simple FF.	45.45
LSTM	28.56
DeepArEst	26.67

Table 4: Prediction models.

state-of-the-art traditional and ML-based prediction models used in prior works [47, 86]. As shown in Algorithm 2, for every model under a periodic scheduling interval of 1 minute (T_s), we use the *Predicted*_load (L_p) at time $T + T_p$ and compare it with the *current_load* to determine the number of instances (I_n). T_p is defined as the average launch time for new instances. (T_s) is set to 1 minute as it is the typical instance provisioning time for EC2 VMs. To calculate (L_p), we sample the arrival rate in adjacent windows of size W over the past S seconds. Using the global arrival rate from all windows, the model predicts (L_p) for T_p time units from T. T_p is set to 10 minutes because it is sufficient time to capture the variations in long-term future. All these parameters are tunable based on the system needs.

Importance Sampling: An important concern in autoscaling is that the model selection policy dynamically determines the models in the ensemble for a given request constraints. Autoscaling the instances equally for every model based on predicted load, would inherently lead to over-provisioned instances for under-used models. To address this concern, we design a weighted autoscaling policy which intelligently auto-scales instances for every pool based on the weights. As shown in Algorithm 2, weights are determined by frequency in which a particular model is chosen for requests (*get_popularity*) with respect to other models in the ensemble.

The weights are multiplied with the predicted load to scale instances *(launch_workers)* for every model pool. We name this as an importance sampling **6b** technique, because the model pools are scaled proportional to their popularity.

5 Implementation and Evaluation

We implemented a prototype of *Cocktail* and deployed it on AWS EC2 [5] platform The details of the implementation are described below. Cocktail is open-sourced at `https://github.com/jashwantraj92/cocktail`

5.1 Cocktail Prototype Implementation

Cocktail is implemented using 10KLOC of `Python`. We designed *Cocktail* as a client-server architecture, where one master VM receives all the incoming requests which are sent to individual model worker VMs.

Master-Worker Architecture: The master node handles the major tasks such as (i) concord model selection policy, (ii) request dispatch to workers VMs as asynchronous future tasks using `Python asyncio` library, and (iii) ensembling the prediction from the worker VMs. Also all VM specific metrics such as current_load, CPU utilization, etc. reside in the master node. It runs on a `C5.16x` [8] large instance to handle these large volume of diverse tasks. Each worker VMs runs a client process to serve its corresponding model. The requests are served as independent parallel threads to ensure timely predictions. We use `Python Sanic` web-server for communication with the master and worker VMs. Each worker VM runs tensorflow-serving [60] to serve the inference requests.

Load Balancer: The master VMs runs a separate thread to monitor the importance sampling of all individual model pools. It keeps track of the number of requests served per model in the past 5 minutes. This information is used for calculating the weights per model for autoscaling decisions. We integrate a `mongodb` [21] database in the master node to maintain all information about procured instances, spot-instance price list, and instance utilization. The load prediction model resides in the master VM which constantly records the arrival rate in adjacent windows. Recall that the details of the prediction were described in Section 4.2.2. The DeepAREst [4] model was trained using `Keras` [22] and `Tensorflow`, over 100 epochs with 2 layers, 32 neurons and a batch-size of 1.

Model Cache: We keep track of the model selected for ensembling on a per request constraint basis. The constraints are defined as `<latency,accuracy>` pair. The queries arriving with similar constraints can read the model cache to avoid re-computation for selecting the models. The model cache is implemented as a hash-map using `Redis` [16] in-memory key-value store for fast access.

Constraint specification: We expose a simple API to developers, where they can specify the type of inference task (e.g., classification) along with the `<latency,accuracy>` constraints. Developers also need to indicate the primary objective between these two constraints. *Cocktail* automatically

Dataset	Application	Classes	Train-set	Test-set
ImageNet [29]	Image	1000	1.2M	50K
CIFAR-100 [50]	Image	100	50K	10K
SST-2 [72]	Text	2	9.6K	1.8K
SemEval [66]	Text	3	50.3K	12.2K

Table 5: Benchmark Applications and datasets.

chooses a set of single or ensemble models required to meet the developer specified constraints.

Discussion: Our accuracy and latency constraints are limited to the measurements from the available pretrained models. Note that changing the models or/and framework would lead to minor deviations. While providing latency and top-1% accuracy of the pretrained models is an offline step in *Cocktail*, we can calculate these values through one-time profiling and use them in the framework. All decisions related to VM autoscaling, bin-packing and load-prediction are reliant on the centralized mongodb database, which can become a potential bottleneck in terms of scalability and consistency. This can be mitigated by using fast distributed solutions like Redis [16] and Zookeeper [46]. The DeepARest model is pre-trained using 60% of the arrival trace. For varying load patterns, the model parameters can be updated by re-training in the background with new arrival rates.

5.2 Evaluation Methodology

We evaluate our prototype implementation on AWS EC2 [8] platforms. Specifically, we use `C5.xlarge, 2xlarge, 4xlarge, 8xlarge` for CPU instances and `p2.xlarge` for GPU instances.

Load Generator: We use different traces which are given as input to the load generator. Firstly, we use real-world request arrival traces from Wikipedia [76], which exhibit typical characteristics of ML inference workloads as it has recurring diurnal patterns. The second trace is production twitter [48] trace which is bursty with unexpected load spikes. We use the first 1 hour sample of both the traces and they are scaled to have an average request rate of 50 req/sec.

Workload: As shown in Table 5 we use image-classification and Sentiment Analysis (text) applications with two datasets each for our evaluation. Sentiment analysis outputs the sentiment of a given sentence as positive negative and (or) neutral. We use 9 different prominently used text-classification models from transformers library [81] (details available in appendix) designed using Google BERT [30] architecture trained on `SST` [72] and `SemEval` [66] dataset. Each request from the load-generator is modelled after a query with specific `<latency,accuracy>` constraints. The queries consist of images or sentences, which are randomly picked from the test dataset. In our experiments, we use five different types of these constraints.

As an example for the `Imagenet` dataset shown in Figure 6, each constraint is a representative of <latency, accuracy> combination offered by single models (shown in Table 1). We use one constraint (blue dots) each from five different regions
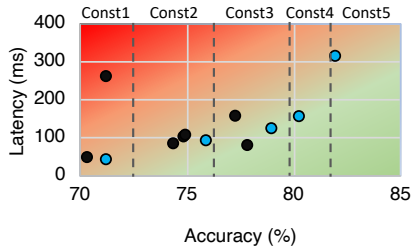
Figure 6: Constraints used in our workloads.

(categorized by dotted lines) picked in the increasing order of accuracy. Each of these picked constraints (named const1 - const5 in the Figure) represents a single baseline model, whose corresponding ensemble size ranges from small (2) to large (10), as shown in Table 3. Note that the latency is the raw model execution latency, and does not include the additional network-transfer overheads incurred. We picked the constraints using a similar procedure by ordering constraints across five different categories for `CIFAR-100`, `SST-2` and `SemEval` (twitter tweets) datasets. The list of models used for them are given in the Appendix. We model two different workload mixes by using a combination of these five query constraint types. Based on the decreasing order of accuracy, we categorize them into *Strict* and *Relaxed* workloads.

5.2.1 Evaluation Metrics

Most of our evaluations of *Cocktail* for image-classification are performed using the `Imagenet` dataset. To further demonstrate the sensitivity of Cocktail to dataset and applicability to other classification applications, we also evaluate it using `CIFAR-100` and Sentiment-Analysis application. We use three important metrics: response latency, cost and accuracy for evaluating and comparing our design to other state-of-the-art systems. The response latency metric includes model inference latency, communication/network latency and synchronization overheads. Queries that do not meet response latency requirements (>700ms) are considered as SLO violations. The cost metric is the billing cost from AWS, and the accuracy metric is measured as the percentage of requests that meet the target accuracy requirements.

We compare these metrics for *Cocktail* against (i) *InFaas* [83], which is our baseline that employs single model selection policy; (ii) *Clipper* [27], which uses static full model selection policy (analogous to AWS AutoGluon); and (iii) *Clipper-X* which is an enhancement to *Clipper* with a simple model selection (drop one model at a time) that does not utilize the *mode*-based policy enforced in *Cocktail*. Both *InFaas* and *Clipper* share *Cocktail*'s implementation setup to ensure a fair comparison with respect to our design and execution environment. For instance, both *Clipper* and *InFaas* employ variants of a reactive autoscaler as described in Section 4.2.2. However, in our setup, both benefit from the distributed autoscaling and prediction policies, thus eliminating variability. Also note that *InFaas* is deployed using OnDemand instances, while both *Clipper* and *Cocktail* use spot instances.

6 Analysis of Results

This section discusses the experimental results of *Cocktail* using the Wiki and Twitter traces. To summarize the overall results, Cocktail providing $2\times$ reduction in latency, while meeting the accuracy for up-to 96% of the requests under reduced deployment cost by $1.4\times$, when compared to *InFaaS* and *Clipper*.

6.1 Latency, Accuracy and Cost Reduction

Latency Distribution: Figure 7 shows the distribution of total response latency in a standard box-and-whisker plot. The boundaries of the box-plots depict the 1st quartile (25th percentile (PCTL)) and 3rd quartile (75th PCTL), the whiskers plot the minimum and maximum (tail) latency and the middle line inside the box depict the median (50 PCTL). The total response latency includes additional 200-300ms incurred for query serialization and data transfer over network. It can be seen that the maximum latency of *Cocktail* is similar to the 75th PCTL latency of *InFaas*. This is because the single model inference have up to 2x higher latency to achieve higher accuracy. Consequently, this leads to 35% SLO violations for *InFaas* in the case of Strict workload. In contrast, both *Cocktail* and *Clipper* can reach the accuracy at lower latency due to ensembling, thus minimizing SLO violations to 1%.

Also, the tail latency is higher for Twitter trace (Figure 7c, 7d) owing to its bursty nature. Note that the tail latency of *Clipper* is still higher than *Cocktail* because *Clipper* ensembles more models than *Cocktail*, thereby resulting in straggler tasks in the VMs. The difference in latency between *Cocktail* and *InFaas* is lower for *Relaxed* workload when compared to *Strict* workload (20% lower in tail). Since the *Relaxed* workload has much lower accuracy constraints, smaller models are able to singularly achieve the accuracy requirements at lower latency.

Accuracy violations: The accuracy is measured as a moving window average with size 200 for all the requests in the workload. Both *Clipper* and *Cocktail* can meet the accuracy for 56% of requests, which is 26% and 9% more than *InFaas* and *Clipper* respectively. This is because, intuitively ensembling leads to higher accuracy than

Scheme	Accuracy Met (%)	
	Strict	Relaxed
InFaas	21	71
Clipper	47	89
Cocktail	56	96

Table 6: Requests meeting target accuracy averaged for both Trace.

single models. However, *Cocktail* is still 9% better than *Clipper* because the class-based weighted voting, is efficient in breaking ties when compared to weighting averaging used in *Clipper*. Since majority voting can include ties in votes, we analyzed the number of ties, which were correctly predicted for all the queries. *Cocktail* was able to deliver correct predictions for 35% of the tied votes, whereas breaking the ties in *Clipper* led only to 20% correct predictions.

(a) Wiki-trace: *Strict* workload. **(b)** Wiki-trace: *Relaxed* workload. **(c)** Twitter-trace: *Strict* workload. **(d)** Twitter-trace: *Relaxed* workload.
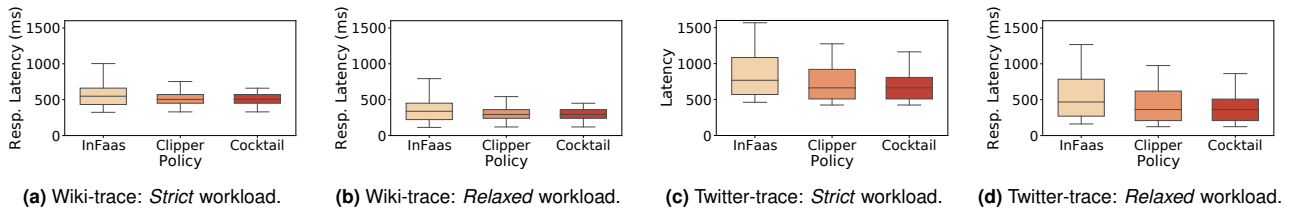
Figure 7: Latency Distribution of *InFaas*, *Clipper* and *Cocktail* for two workload mixes using both Wiki and Twitter traces.
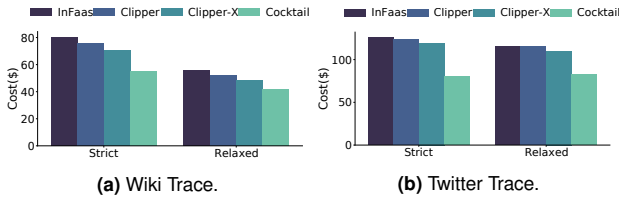

(a) Wiki Trace. **(b)** Twitter Trace.

Figure 8: Cost savings of *Cocktail* compared to three schemes.

(a) Average number of models used in the ensemble. **(b)** Distribution of requests served by each individual model.
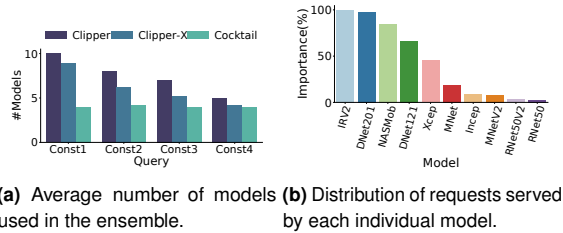
Figure 9: Benefits of dynamic model selection policy.

Note that, changing the target accuracy to tolerate a 0.5% loss, increases the percentage of requests that meet accuracy to 81% for *Cocktail*, when compared to 61% for *InFaas*. The requests meeting accuracy are generally higher for the *Relaxed* workload because the target accuracy is much lower. Overall, *Cocktail* was able to deliver an accuracy of 83% and 79.5% on average for the *Strict* and *Relaxed* workloads, respectively. This translates to 1.5% and 1% better accuracy than *Clipper* and *InFaas*. We do not plot the results for *Clipper-X*, which achieves similar accuracy to *Cocktail*, but uses more models as explained in Section 6.2.1.

Cost Comparison: Figure 8 plots the cost savings of *Cocktail* when compared to *InFaas*, *Clipper* and *Clipper-X* policies. It can be seen that, *Cocktail* is up to 1.45× more cost effective than *InFaas* for *Strict* workload. In addition, *Cocktail* reduces cost by 1.35× and 1.27× compared to *Clipper* and *Clipper-X* policies, owing to its dynamic model selection policy, which minimizes the resource footprint of ensembling. On the other hand, *Clipper* uses all models in ensemble and the *Clipper-X* policy does not right size the models as aggressively as *Clipper*, hence they are more expensive. Note that, all the schemes incur higher cost for twitter trace (Figure 8b) compared to wiki trace (Figure 8a). This is because the twitter workload is bursty, thereby leading to intermittent over-provisioned VMs.

6.2 Key Sources of Improvements

The major improvements in terms of cost, latency, and accu-

racy in *Cocktail* are explained below. For brevity in explanation, the results are averaged across Wiki and Twitter traces for strict workload.

6.2.1 Benefits from dynamic model selection

Figure 9a plots the average number of models used for queries falling under the first four different constraint (const) types. Here, *Cocktail* reduces the number of models by up to 55% for all four query types. This is because our dynamic policy ensures that the number of models are well within $N/2$ most of the time, whereas the *Clipper-X* policy does not aggressively scale down models. *Clipper*, on the other hand, is static and always uses all the models. The percentage of model-reduction is lower for *Const2*, 3 and 4 because, the total models used in the ensemble is less than *Const1* (8, 7 and 6 models, respectively). Still, the savings in terms of cost will be significant because even removing one model from the ensemble amounts to ∼20% cost savings in the long run (*Clipper* vs *Clipper-X* ensemble in Figure 8). Thus, the benefits of *Cocktail* are substantial for large ensembles while reducing the number of models for medium-sized ensembles.

Figure 9b shows the breakdown of the percentage of requests (*Const1*) served by the each model. As seen, InceptionResNetV2, Densenet-201, Densenet121, NasnetMobile and Xception are the top-5 most used models in the ensemble. Based on Table 1, if we had statically taken the top $N/2$ most accurate models, NasNetmobile would not have been included in the ensemble. However, based on the input images sent in each query, our model selection policy has been able to identify NasNetMobile to be a significantly contributing model in the ensemble. Further, the other 5 models are used by up to 25% of the images. Not including them in the ensemble would have led to severe loss in accuracy. But, our dynamic policy with the class-based weighted voting, adapts to input images in a given interval by accurately selecting the best performing model for each class. To further demonstrate the effectiveness of our dynamic model selection,

Figure 10b,10c plots the number models in every sampling interval along with cumulative accuracy and window accuracy within each sampling interval for three schemes. We observe that *Cocktail* can effectively scale up and scale down the models while maintaining the cumulative accuracy well within the threshold. More than 50% of the time the number of models are maintained between 4 to 5, because the dynamic policy is quick in detecting accuracy failures and recovers immediately
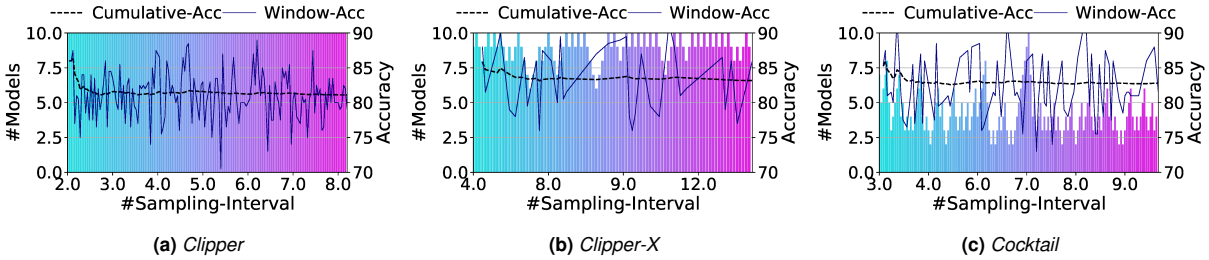
(a) *Clipper* **(b)** *Clipper-X* **(c)** *Cocktail*

Figure 10: Figures (a), (b) and (c) shows the number of models used in ensemble with corresponding cumulative accuracy and window accuracy over a 1 hour period for requests under *Const1*. Figure (d) shows the effects of distributed autoscaling with importance sampling.
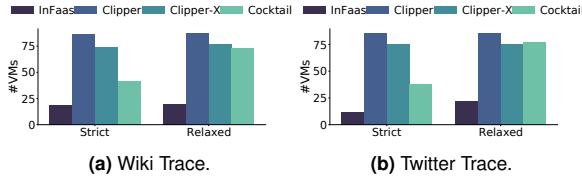

(a) Wiki Trace. **(b)** Twitter Trace.

Figure 11: Number of VMs spawned for all four schemes.

(a) Cumulative #VMs. **(b)** Failure Analysis.
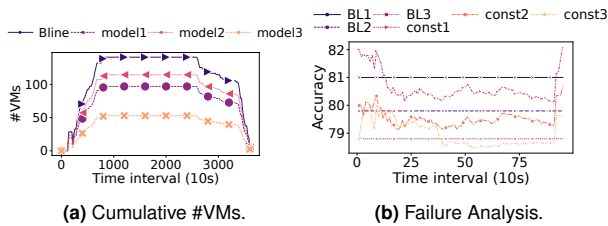
Figure 12: Sensitivity analysis of VMs.

by scaling up models. However, *Clipper-X* does not scale down models as frequently as *Cocktail*, while ensuring similar accuracy. *Clipper* is less accurate than *Cocktail* and further it uses all 10 models throughout.

6.2.2 Benefits from Autoscaling

Figure 11 plots the reduction in the number of VMs used by all four schemes. It can be seen that both *Cocktail* and *Clipper-X* spawn 49% and 20% fewer VMs than *Clipper* for workload-1 on Twitter trace. *Cocktail* spawns 29% lesser VMs on top of *Clipper-X*, because it is not aggressive enough like *Cocktail* to downscale more models at every interval. It is to be noted that the savings are lower for *Relaxed* workload because, the number of models in the ensemble are inherently low, thus leading to reduced benefits from scaling down the models. Intuitively, *InFaas* has the least number of VMs spawned because it does not ensemble models. *Cocktail* spawns upto 50% more VMs than *InFaas*, but in turns reduces accuracy loss by up to 96%.

To further capture the benefits of the weighted autoscaling policy, Figure 12a plots the number of VMs spawned over time for the top-3 most used models in the ensemble for *Const1*. The Bline denotes number of VMs that would be spawned without applying the weights. Not adopting an importance sampling based weighted policy would result in equivalent number of VMs as the Bline for all models. However, since *Cocktail* exploits importance sampling by keeping track of the frequency in which models are selected, the num-

ber of VMs spawned for model1, model2 and model-3 is upto $3\times$ times lesser than uniform scaling. Figure 9b shows the most used models in decreasing order of importance. The autoscaling policy effectively utilizes this importance factor in regular intervals of 5 minutes. Despite using multiple models for a single inference, importance sampling combined with aggressive model pruning, greatly reduces the resource footprint which directly translates to the cost savings in *Cocktail*.

6.2.3 Benefits of Transient VMs

The cost-reductions in *Cocktail* are akin to cost-savings of transient VMs compared to On-Demand (OD) VMs. We profile the spot price of 4 types of C5 EC2 VMs over a 2-week period in August 2020. It was seen that, the spot instance prices have predictable fluctuations. When compared to the OD price , they were up to 70% cheaper. This price gap is capitalized in *Cocktail* to reduce the cost of instances consumed by ensembling. Note that, we set the bidding price conservatively to 40% of OD. Although, *Cocktail* spawns about 50% more VMs than *InFaas*, the high P_f of small models and spot-instance price reductions combined with autoscaling policies lead to the overall 30-40% cost savings.

6.3 Sensitivity Analysis

In this section, we analyze the sensitivity of *Cocktail* with respect to various design choices which include (i) sampling interval of the accuracy measurements, (ii) spot-instance failure rate and (iii) type of datasets and applications.

6.3.1 Sampling Interval

To study the sensitivity with respect to the sampling interval for measure accuracy loss/gain, we use four different intervals of 10s, 30s, 60s and 120s. Figure 13 plots the average number of models (bar- left y-axis) and cumulative accuracy (line-right y-axis) for the different sampling intervals for queries with three different constraints. It can be seen that the 30s interval strikes the right balance with less than 0.2% loss in accuracy and has average number models much lesser than other intervals. This is because, increasing the interval leads to lower number of scale down operations, thus resulting in a bigger ensemble. As a result, the 120s interval has the highest number of models.
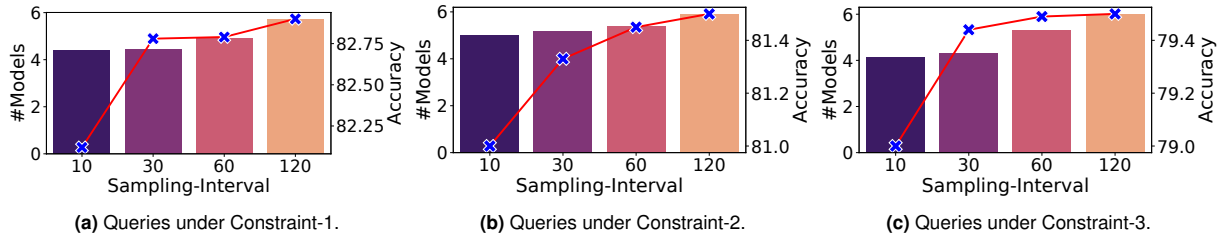
(a) Queries under Constraint-1. **(b)** Queries under Constraint-2. **(c)** Queries under Constraint-3.

Figure 13: Sensitivity analysis of model selection with respect to sampling interval. The average number of models is in primary axis and cumulative accuracy in secondary axis.

6.3.2 Cocktail Failure Resilience

We use spot instances to host models in *Cocktail*. As previously discussed in Section 3, spot instances interruptions can lead to intermittent loss in accuracy as certain models will be unavailable in the ensemble. However for large ensembles (5 models are more), the intermittent accuracy loss is very low. Figure 12b plots the failure analysis results for top three constraints by comparing the ensemble accuracy to the target accuracy. The desired accuracy for all three constraints are plotted as BL1, BL2 and BL3. We induce failures in the instances using *chaosmonkey* [19] tool with a 20% failure probability. It can be seen that queries in all three constraints suffer an intermittent loss in accuracy of 0.6% between the time period 240s and 800s. Beyond 800s, they quickly recover back to the required accuracy because additional instances are spawned in place of failed instances. However, in the case of *InFaas*, this would lead to 1% failed requests due to requests being dropped from the failed instances.

An alternate solution would be to restart the queries in running instances but that leads to increased latencies for the 1% requests. In contrast, *Cocktail* incurs a modest accuracy loss of well within 0.6% and quickly adapts to reach the target accuracy. Thus, *Cocktail* is inherently fault-tolerant owing to the parallel nature in computing multiple inferences for a single request. We observe similar accuracy loss or lower for different probability failures of 5%, 10% and 25%, respectively (results/charts omitted in the interest of space).

Discussion: For applications that are latency tolerant, we can potentially redirect requests from failed instances to existing instances, which would lead to increased tail latency. The results we how are only for latency intolerant applications. Note that, the ensembles used in our experiments are at-least 4 models or more. For smaller ensembles, instance failures might lead to higher accuracy loss, but in our experiments, single models typically satisfy their constraints.

6.3.3 Sensitivity to Constraints

Figure 14 plots the sensitivity of model selection policy under a wide-range of latency and accuracy constraints. In Figure 14a, we vary the latency under six different constant accuracy categories. It can be seen that for fixed accuracy of 72%, 78% and 80%, the average number of models increase with increase in latency, but drops to 1 for the highest latency. Intuitively, singe large models with higher latency can satisfy
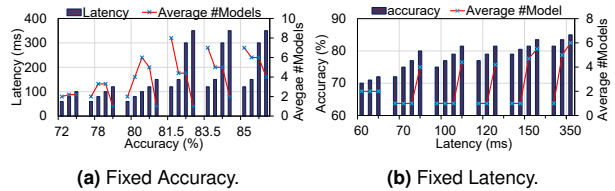

(a) Fixed Accuracy. **(b)** Fixed Latency.

Figure 14: Sensitivity Constraints under fixed latency and accuracy. Bar graphs (latency) plotted using primary y-axis and line graph (#models) plotted using secondary y-axis.
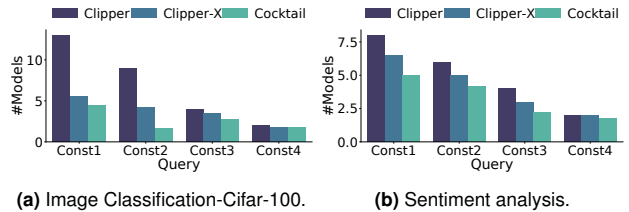

(a) Image Classification-Cifar-100. **(b)** Sentiment analysis.

Figure 15: Average number of models used in the ensemble.

the accuracy, while short latency models need to be ensembled to reach the same accuracy. For accuracy greater than 80%, the ensemble size drops with higher latencies. This is because the models which offer higher accuracy are typically dense and hence, smaller ensembles are sufficient. In Figure 14b, we vary the accuracy under six different constant latency categories. It can be seen that for higher accuracies, *Cocktail* tries to ensemble more models to reach the accuracy, while for lower accuracy it resorts to using single models.

6.3.4 Sensitivity to Dataset

To demonstrate the applicability of *Cocktail* to multiple datasets, we conducted similar experiments as elucidated in Section 5.2.1 using the CIFAR-100 dataset [50]. It comprises of 100 distinct image classes and we trained 11 different models including the nine that are common from Table 1. Figure 15a plots the average number of models used by the three policies for the top four constraints. It can be seen that *Cocktail* shows similar reduction (as Imagenet) while using only 4.4 models on average. As expected, *Clipper* and *Clipper-X* use more models than *Cocktail* (11 and 5.4, respectively) due to non-aggressive scaling down of the models used.

Figure 16a plots the latency reduction and accuracy boost when compared to *InFaaS* (baseline). While able to reduce 60% of the models used in the ensemble, *Cocktail* also reduces latency by up to 50% and boosts accuracy by up to 1.2%. *Cocktail* was also able to deliver modest accuracy gain
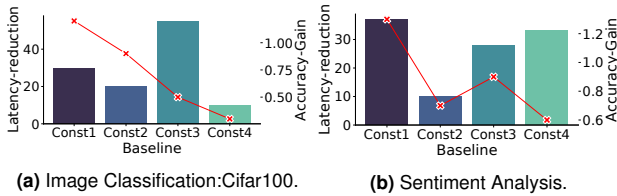
(a) Image Classification:Cifar100.

(b) Sentiment Analysis.

Figure 16: Latency reduction (%) plotted as bar graph(primary y-axis) and accuracy gains (%) plotted as line graph (secondary y-axis) over InFaaS.
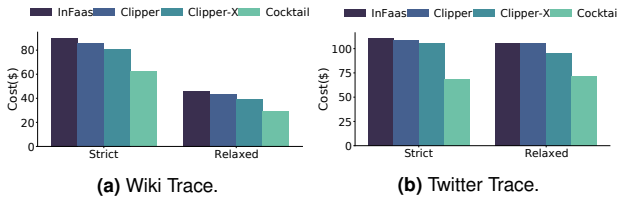

(a) Wiki Trace.

(b) Twitter Trace.

Figure 17: Cost savings of Cocktail for Sentiment Analysis.

of 0.5% than *Clipper* (not plotted). The accuracy gain seen in `CIFAR-100` is lesser than ImageNet dataset because the class-based weighted voting works effectively when handling large number of classes (100 in `CIFAR` vs 1000 in ImageNet). Nevertheless, *Cocktail* is able to deliver the accuracy at 2x lower latency than *InFaaS* and 1.35x lower cost than Clipper.

6.4 General Applicability of Cocktail

To demonstrate the general applicability of *Cocktail* to other classification tasks, we evaluated *Cocktail* using a Sentiment Analysis application for two datasets. The results reported are averaged across both the datasets. Figure 15b plots the average number of models used by the three policies for the top four constraints. As shown for *Const1*, *Cocktail* shows similar reduction (as image-classification) with only using 4.8 models on average, which is 40% and 26% lower than *Clipper* and *Clipper-X*, respectively. *Cocktail* is also able to reduce the number of models by 30% and 50% for medium ensembles (*Const2* & *Const3*) as well.

Figure 16b plots the latency reduction and accuracy gain, compared to *InFaaS* (baseline). While being able to reduce 50% of the models used in the ensemble, *Cocktail* also reduces latency by up to 50% and improves accuracy by up to 1.3%. Both *Cocktail* and *Clipper* deliver the same overall accuracy (96%, 94.5%, 93.5%, and 92%)). Since sentiment analysis only has 2-3 classes, there are no additional accuracy gains by using the class-based weighted voting. However, the model selection policy effectively switches between different models based on the structure of input text (equivalent to classes in images). For instance, complex sentences are more accurately classified by denser models compared to smaller. Despite the lower accuracy gains, *Cocktail* is able to reduce the cost (Figure 17) of model-serving by $1.45\times$ and $1.37\times$ for Wiki trace compared to *InFaaS* and *Clipper*, respectively.

7 Concluding Remarks

There is an imminent need to develop model serving systems that can deliver highly accurate, low latency predictions at reduced cost. In this paper, we propose and evaluate *Cocktail*, a cost-effective model serving system that exploits ensembling techniques to meet high accuracy under low latency goals. In *Cocktail*, we adopt a three-fold approach to reduce the resource footprint of model ensembling. More specifically, we (i) develop a novel dynamic model selection, (ii) design a prudent resource management scheme that utilizes weighted autoscaling for efficient resource allocation, and (iii) leverage transient VM instances to reduce the deployment costs. Our results from extensive evaluations using both CPU and GPU instances on AWS EC2 cloud platform demonstrate that *Cocktail* can reduce deployment cost by $1.4\times$, while reducing latency by $2\times$ and satisfying accuracy for 96% of requests, compared to the state-of-the-art model-serving systems.

Acknowledgments

References

[1] Martín Abadi. Tensorflow: learning functions at scale. In *Acm Sigplan Notices*. ACM, 2016.

[2] Deepak Agarwal, Bo Long, Jonathan Traupman, Doris Xin, and Liang Zhang. Laser: A scalable response prediction platform for online advertising. In *Proceedings of the 7th ACM international conference on Web search and data mining*, pages 173–182, 2014.

[3] Ahmed Ali-Eldin, Jonathan Westin, Bin Wang, Prateek Sharma, and Prashant Shenoy. Spotweb: Running latency-sensitive distributed web services on transient cloud servers. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, pages 1–12, 2019.

[4] Amazon. Deepar estimator. https://docs.aws.amazon.com/sagemaker/latest/dg/deepar.html,February2020.

[5] Amazon. EC2 pricing. https://aws.amazon.com/ec2/pricing/.

[6] Amazon. Sagemaker. https://aws.amazon.com/sagemaker/, February 2018.

[7] Amazon. Azure Low priority batch VMs., February 2018. https://docs.microsoft.com/en-us/azure/batch/batch-low-pri-vms .

[8] Amazon. EC2 C5 Instances., February 2018. https://aws.amazon.com/ec2/instance-types/c5/ .

[9] Amazon. Google Preemptible VMs., February 2018. https://cloud.google.com/preemptible-vms .

[10] Azure. Machine Learning as a Service., February 2018. https://azure.microsoft.com/en-us/pricing/details/machine-learning-service/ .

[11] Azure. Ensembling in Azure ML Studio., February 2020. https://docs.microsoft.com/en-us/azure/machine-learning/studio-module-reference/multiclass-decision-forest .

[12] Ataollah Fatahi Baarzi, Timothy Zhu, and Bhuvan Urgaonkar. Burscale: Using burstable instances for cost-effective autoscaling in the public cloud. In *Proceedings of the ACM Symposium on Cloud Computing*, New York, NY, USA, 2019. Association for Computing Machinery.

[13] Marian Stewart Bartlett, Gwen Littlewort, Mark Frank, Claudia Lainscsek, Ian Fasel, and Javier Movellan. Recognizing facial expression: machine learning and application to spontaneous behavior. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 2, pages 568–573. IEEE, 2005.

[14] Eric Bauer and Ron Kohavi. An empirical comparison of voting classification algorithms: Bagging, boosting, and variants. *Machine learning*, 36(1-2):105–139, 1999.

[15] William H Beluch, Tim Genewein, Andreas Nürnberger, and Jan M Köhler. The power of ensembles for active learning in image classification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 9368–9377, 2018.

[16] Josiah L Carlson. *Redis in action*. Manning Publications Co., 2013.

[17] Rich Caruana, Alexandru Niculescu-Mizil, Geoff Crew, and Alex Ksikes. Ensemble selection from libraries of models. In *Proceedings of the twenty-first international conference on Machine learning*, page 18, 2004.

[18] Jesús Cerquides and Ramon López De Mántaras. Robust bayesian linear classifier ensembles. In *European Conference on Machine Learning*, pages 72–83. Springer, 2005.

[19] Michael Alan Chang, Bredan Tschaen, Theophilus Benson, and Laurent Vanbever. Chaos monkey: Increasing sdn reliability through systematic network destruction. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 371–372, 2015.

[20] Lingjiao Chen, Matei Zaharia, and James Zou. Frugalml: How to use ml prediction apis more accurately and cheaply. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

[21] Kristina Chodorow. *MongoDB: the definitive guide: powerful and scalable data storage*. " O'Reilly Media, Inc.", 2013.

[22] Francois Chollet. *Deep Learning mit Python und Keras: Das Praxis-Handbuch vom Entwickler der Keras-Bibliothek*. MITP-Verlags GmbH & Co. KG, 2018.

[23] Andrew Chung, Jun Woo Park, and Gregory R. Ganger. Stratus: Cost-aware container scheduling in the public cloud. In *SoCC*, 2018.

[24] Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM conference on recommender systems*, pages 191–198, 2016.

[25] Daniel Crankshaw, Peter Bailis, Joseph E Gonzalez, Haoyuan Li, Zhao Zhang, Michael J Franklin, Ali Ghodsi, and Michael I Jordan. The missing piece in complex analytics: Low latency, scalable model management and serving with velox. *arXiv preprint arXiv:1409.3809*, 2014.

[26] Daniel Crankshaw, Gur-Eyal Sela, Corey Zumar, Xiangxi Mo, Joseph E. Gonzalez, Ion Stoica, and Alexey Tumanov. Inferline: ML inference pipeline composition framework. *CoRR*, abs/1812.01776, 2018.

[27] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, Boston, MA, March 2017. USENIX Association.

[28] Deepstudio. Deep Learning Dtudio, February 2020. https://docs.deepcognition.ai/ .

[29] J. Deng, W. Dong, R. Socher, L. Li, and and. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, June 2009.

[30] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[31] Nick Erickson, Jonas Mueller, Alexander Shirkov, Hang Zhang, Pedro Larroy, Mu Li, and Alexander Smola. Autogluon-tabular: Robust and accurate automl for structured data, 2020.

[32] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[33] Mikel Galar, Alberto Fernandez, Edurne Barrenechea, Humberto Bustince, and Francisco Herrera. A review on ensembles for the class imbalance problem: Bagging-, boosting-, and hybrid-based approaches. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(4):463–484, 2012.

[34] Arpan Gujarati, Sameh Elnikety, Yuxiong He, Kathryn S. McKinley, and Björn B. Brandenburg. Swayam: Distributed Autoscaling to Meet SLAs of Machine Learning Inference Services with Resource Efficiency. In *USENIX Middleware Conference*, 2017.

[35] Arpan Gujarati, Reza Karimi, Safya Alzayat, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving dnns like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, Banff, Alberta, November 2020. USENIX Association.

[36] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Nachiappan C.Nachiappan, Mahmut Taylan Kandemir, and Chita R. Das. Fifer: Tackling Resource Underutilization in the Serverless Era. In *USENIX Middleware Conference*, 2020.

[37] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Mahmut Taylan Kandemir, Bhuvan Urgaonkar, George Kesidis, and Chita Das. Spock: Exploiting serverless functions for slo and cost aware resource procurement in public cloud. In *IEEE CLOUD*, 2019.

[38] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Cyan Subhra Mishra, Mahmut Taylan Kandemir, and Chita R. Das. Towards designing a self-managed machine learning inference serving system inpublic cloud, 2020.

[39] U. Gupta, S. Hsia, V. Saraph, X. Wang, B. Reagen, G. Wei, H. S. Lee, D. Brooks, and C. Wu. Deeprecsys: A system for optimizing end-to-end at-scale neural recommendation inference. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 982–995, 2020.

[40] Rui Han, Moustafa M. Ghanem, Li Guo, Yike Guo, and Michelle Osmond. Enabling cost-aware and adaptive elasticity of multi-tier cloud applications. *Future Gener. Comput. Syst.*, 32(C):82–98, March 2014.

[41] Aaron Harlap, Andrew Chung, Alexey Tumanov, Gregory R. Ganger, and Phillip B. Gibbons. Tributary: spot-dancing for elastic services with latency SLOs. In *ATC*, 2018.

[42] Aaron Harlap, Alexey Tumanov, Andrew Chung, Gregory R. Ganger, and Phillip B. Gibbons. Proteus: Agile ML Elasticity Through Tiered Reliability in Dynamic Resource Markets. In *Eurosys*, 2017.

[43] Johann Hauswald, Michael A. Laurenzano, Yunqi Zhang, Cheng Li, Austin Rovinski, Arjun Khurana, Ronald G. Dreslinski, Trevor Mudge, Vinicius Petrucci, Lingjia Tang, and Jason Mars. Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers. In *ASPLOS*, 2015.

[44] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang. Applied machine learning at facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 620–629, Feb 2018.

[45] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. Searching for mobilenetv3. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1314–1324, 2019.

[46] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, 2010.

[47] Minoru Kawashima, Charles E Dorgan, and John W Mitchell. Hourly thermal load prediction for the next 24 hours by arima, ewma, lr and

an artificial neural network. Technical report, American Society of Heating, Refrigerating and Air-Conditioning Engineers . . . , 1995.

[48] Abeer Abdel Khaleq and Ilkyeun Ra. Cloud-based disaster management as a service: A microservice approach for hurricane twitter data analysis. In *GHTC*, 2018.

[49] J Zico Kolter and Marcus A Maloof. Dynamic weighted majority: An ensemble method for drifting concepts. *Journal of Machine Learning Research*, 8(Dec):2755–2790, 2007.

[50] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-100 (canadian institute for advanced research), 2010. http://www.cs.toronto.edu/~kriz/cifar.html.

[51] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942*, 2019.

[52] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. PRETZEL: Opening the black box of machine learning prediction serving systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 611–626, Carlsbad, CA, October 2018. USENIX Association.

[53] Romain Lerallut, Diane Gasselin, and Nicolas Le Roux. Large-scale real-time product recommendation at criteo. In *Proceedings of the 9th ACM Conference on Recommender Systems*, pages 232–232, 2015.

[54] Weibo Liu, Zidong Wang, Xiaohui Liu, Nianyin Zeng, Yurong Liu, and Fuad E Alsaadi. A survey of deep neural network architectures and their applications. *Neurocomputing*, 234:11–26, 2017.

[55] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[56] Zhenyu Lu, Xindong Wu, Xingquan Zhu, and Josh Bongard. Ensemble pruning via individual contribution ordering. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '10, page 871–880, New York, NY, USA, 2010. Association for Computing Machinery.

[57] Cyan Subhra Mishra, Jack Sampson, Mahmut Taylan Kandemir, and Vijaykrishnan Narayanan. Origin: Enabling on-device intelligence for human activity recognition using energy harvesting wireless sensor networks. In *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1414–1419, 2021.

[58] Soo-Jin Moon, Jeffrey Helt, Yifei Yuan, Yves Bieri, Sujata Banerjee, Vyas Sekar, Wenfei Wu, Mihalis Yannakakis, and Ying Zhang. Alembic: Automated model inference for stateful network functions. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 699–718, Boston, MA, February 2019. USENIX Association.

[59] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[60] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. Tensorflow-serving: Flexible, high-performance ml serving. *arXiv preprint arXiv:1712.06139*, 2017.

[61] Nikunj C Oza. Online bagging and boosting. In *2005 IEEE international conference on systems, man and cybernetics*, volume 3, pages 2340–2345. Ieee, 2005.

[62] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pages 8026–8037, 2019.

[63] Heyang Qin, Syed Zawad, Yanqi Zhou, Lei Yang, Dongfang Zhao, and Feng Yan. Swift machine learning model serving scheduling: a region

based reinforcement learning approach. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–23, 2019.

[64] Xueheng Qiu, Le Zhang, Ye Ren, Ponnuthurai N Suganthan, and Gehan Amaratunga. Ensemble deep learning for regression and time series forecasting. In *2014 IEEE symposium on computational intelligence in ensemble learning (CIEL)*, pages 1–6. IEEE, 2014.

[65] Atul Rahman, Jongeun Lee, and Kiyoung Choi. Efficient fpga acceleration of convolutional neural networks using logical-3d compute array. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1393–1398. IEEE, 2016.

[66] Sara Rosenthal, Noura Farra, and Preslav Nakov. SemEval-2017 task 4: Sentiment analysis in Twitter. In *Proceedings of the 11th International Workshop on Semantic Evaluation (SemEval-2017)*, pages 502–518, Vancouver, Canada, August 2017. Association for Computational Linguistics.

[67] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.

[68] Prateek Sharma, David Irwin, and Prashant Shenoy. Portfolio-driven resource management for transient cloud servers. *Proc. ACM Meas. Anal. Comput. Syst.*, 1(1), June 2017.

[69] Prateek Sharma, Stephen Lee, Tian Guo, David Irwin, and Prashant Shenoy. Spotcheck: Designing a derivative iaas cloud on the spot market. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–15, 2015.

[70] Steven A Shaya, Neal Matheson, John Anthony Singarayar, Nikiforos Kollias, and Jeffrey Adam Bloom. Intelligent performance-based product recommendation system, October 5 2010. US Patent 7,809,601.

[71] Richard Socher, Yoshua Bengio, and Chris Manning. Deep learning for nlp. *Tutorial at Association of Computational Logistics (ACL)*, 2012.

[72] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1631–1642, 2013.

[73] Mingxing Tan and Quoc V Le. Efficientnet: Rethinking model scaling for convolutional neural networks. *arXiv preprint arXiv:1905.11946*, 2019.

[74] P. Thinakaran, J. R. Gunasekaran, B. Sharma, M. T. Kandemir, and C. R. Das. Phoenix: A constraint-aware scheduler for heterogeneous datacenters. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, June 2017.

[75] P. Thinakaran, J. R. Gunasekaran, B. Sharma, M. T. Kandemir, and C. R. Das. Kube-Knots: Resource Harvesting through Dynamic Container Orchestration in GPU-based Datacenters. In *CLUSTER*, 2019.

[76] Guido Urdaneta, Guillaume Pierre, and Maarten Van Steen. Wikipedia workload analysis for decentralized hosting. *Computer Networks*, 2009.

[77] Alexander Vezhnevets and Vladimir Vezhnevets. Modest adaboost-teaching adaboost to generalize better. In *Graphicon*, pages 987–997, 2005.

[78] Jasper A Vrugt and Bruce A Robinson. Treatment of uncertainty using ensemble methods: Comparison of sequential data assimilation and bayesian model averaging. *Water Resources Research*, 43(1), 2007.

[79] Cheng Wang, Bhuvan Urgaonkar, Neda Nasiriani, and George Kesidis. Using burstable instances in the public cloud: Why, when and how? *SIGMETRICS*, June 2017.

[80] Wei Wang, Jinyang Gao, Meihui Zhang, Sheng Wang, Gang Chen, Teck Khim Ng, Beng Chin Ooi, Jie Shao, and Moaz Reyad. Rafiki: machine learning as an analytics service system. *Proceedings of the VLDB Endowment*, 12(2):128–140, 2018.

[81] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. Huggingface's transformers: State-of-

the-art natural language processing. *arXiv preprint arXiv:1910.03771*, 2019.

[82] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, et al. Machine learning at facebook: Understanding inference at the edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 331–344. IEEE, 2019.

[83] Neeraja J. Yadwadkar, Francisco Romero, Qian Li, and Christos Kozyrakis. A case for managed and model-less inference serving. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, New York, NY, USA, 2019. Association for Computing Machinery.

[84] Tien-Ju Yang, Andrew G. Howard, Bo Chen, Xiao Zhang, Alec Go, Vivienne Sze, and Hartwig Adam. Netadapt: Platform-aware neural network adaptation for mobile applications. *CoRR*, abs/1804.03230, 2018.

[85] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. *arXiv preprint arXiv:1906.08237*, 2019.

[86] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving. In *ATC*, 2019.

[87] Honglei Zhuang, Chi Wang, and Yifan Wang. Identifying outlier arms in multi-armed bandit. In *Advances in Neural Information Processing Systems*, pages 5204–5213, 2017.

[88] Sheikh Ziauddin and Matthew N Dailey. Iris recognition performance enhancement using weighted majority voting. In *2008 15th IEEE International Conference on Image Processing*, pages 277–280. IEEE, 2008.

Appendix

A Modeling of Ensembling

While performing an ensemble it is important to be sure that we can reach the desired accuracy by combining more models. In our design, we solve our first objective function (described in Section 4.1) by combining all available models which meet the latency SLO. To be sure that the combination will give us the desired accuracy of the larger model, we try to theoretically analyse the scenario. We formulate the problem conservatively as following.

We perform an inference by ensembling 'N' models, and each of these models have accuracy 'a'. Therefore the probability of any model giving a correct classification is 'a'. We assume the output to be correct if majority of them, i.e. $\lfloor N/2 \rfloor + 1$ of them give the same result. Then, the final accuracy of this ensemble would be the probability of at least $\lfloor N/2 \rfloor + 1$ of them giving a correct result.

To we model this problem as a coin-toss problem involving N biased coins with having probability of occurrence of head to be a. Relating this to our problem, each coin represents a model, and an occurrence of head represents the model giving the correct classification. Hence, the problem boils down to find the probability of at least $\lfloor N/2 \rfloor + 1$ heads when all N coins are tossed together. This is a standard binomial distribution problem and can be solved by using the following formula:

$$P_{head} = \sum_{i=\lfloor \frac{N}{2} \rfloor + 1}^{N} \binom{N}{i} a^i (1-a)^{(N-i)}.$$

To further quantify, let us consider the case where we need to determine if we can reach the accuracy of NasNetLarge (82%) by combining rest of the smaller models which have lesser latency than NasNetLarge. We have 10 (therefore N = 10) such models and among them the least accurate model is MobileNetV1 (accuracy 70%, therefore a = 0.70). We need to find the probability of at least 6 of them being correct. Using the equation above we find the probability to be

$$P_{head} = \sum_{i=\lfloor \frac{10}{2} \rfloor + 1 = 6}^{10} \binom{10}{i} 0.7^i (1-0.7)^{(10-i)} = 0.83$$

This corresponds to an accuracy of 83%, which is greater than our required accuracy of 82%). Given all the other models have higher accuracy, the least accuracy we can expect with such an ensemble is 83%. This analysis forms the base of our ensemble technique, and hence proving the combination of multiple available models can be more accurate than the most accurate individual model.

B Why DeepARest Model?

We quantitatively justify the choice of using DeepARest by conducting a brick-by-brick comparison of the accuracy loss

when compared with other state-of-the-art prediction models used in prior work.

Table 4 shows the root mean squared error (RMSE) incurred by all the models. The ML models used in these experiments are pre-trained with 60% of the Twitter arrival trace. It is evident that the LSTM and DeepAREst have lowest RMSE value. DeepARest is 10% better than LSTM model. Since the primary contribution in *Cocktail* is to provide high accuracy and low latency predictions at cheaper cost, application developers can adapt the prediction algorithm to their needs or even plug-in their own prediction models.

C System Overheads

We characterize the system-level overheads incurred due to the design choices in *Cocktail*. The *mongodb* database is a centralized server, which resides on the head-node. We measure the overall average latency incurred due to all reads/writes in the database, which is well within 1.5ms. The DeepAREst prediction model which is not in the critical decision-making path runs as a background process incurring 2.2 ms latency on average. The weighted majority voting takes 0.5ms and the model selection policy takes 0.7ms. The time taken to spawn new VM takes about 60s to 100s depending on the size of the VM instance. The time taken to choose models from the model-cache is less than 1ms. The end-to-end response time to send the image to a worker VM and get the prediction back, was dominated by about 300ms (at maximum) of payload transfer time.

D Instance configuration and Pricing

Instance	vCPUs	Memory	Price
C5a.xlarge	4	8 GiB	$0.154
C5a.2xlarge	8	16 GiB	$0.308
C5a.4xlarge	16	32 GiB	$0.616
C5a.8xlarge	32	64 GiB	$1.232

Table 7: Configuration and Pricing for EC2 C5 instances.

E CIFAR-100 and BERT Models

Table 8 shows the different models available for image prediction, that are pretrained on Keras using `CIFAR-100` dataset.

Model	Params (M)	Top-1 Accuracy(%)	Latency (ms)	P_f
Albert-base [51]	11	91.4	55	7
CodeBert [32]	125	89	79	6
DistilBert [67]	66	90.6	92	5
Albert-large	17	92.5	120	4
XLNet [85]	110	94.6	165	3
Bert [30]	110	92	185	3
Roberta [55]	355	94.3	200	2
Albert-xlarge	58	93.8	220	1
Albert-xxlarge	223	95.9	350	1

Table 9: Pretrained models for Sentiment Analysis using BERT.

Similarly Table 9 shows the different models trained for BERT-based sentiment analysis on twitter dataset.

Model	Params (M)	Top1 Accuracy %	Latency (ms)	Pf
Squeezenet	4,253,864	70.10	43.45	10
MobileNEt V2	4,253,864	68.20	41.5	10
Inception V4	23,851,784	76.74	74	6
Resnet50	95,154,159	79.20	98.22	5
ResNet18	44,964,665	76.26	35	6
DenseNet-201	20,242,984	79.80	152.21	2
DenseNet-121	8,062,504	78.72	102.35	3
Xxception	22,910,480	77.80	119.2	4
NasNet	5,326,716	77.90	120	3
InceptionResnetV2	2,510,000	80.30	251.96	1

Table 8: Pretrained models for CIFAR-100 using Imagenet.

F Spot Instance Price Variation

We profile the spot price of 4 types of `C5` EC2 VMs over a 2-week period in August 2020. The price variation is shown in Fig18.
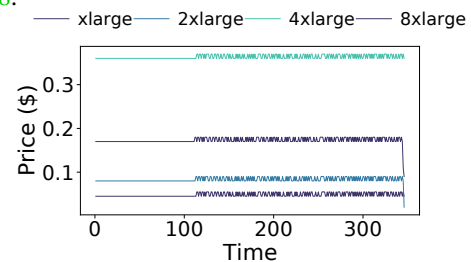

Figure 18: Spot instance price variation (time is in hours).

Data-Parallel Actors:
A Programming Model for Scalable Query Serving Systems

Peter Kraft
Stanford University

Fiodar Kazhamiaka
Stanford University

Peter Bailis
Stanford University

Matei Zaharia
Stanford University

Abstract

We present *data-parallel actors (DPA)*, a programming model for building distributed query serving systems. Query serving systems are an important class of applications characterized by low-latency data-parallel queries and frequent bulk data updates; they include data analytics systems like Apache Druid, full-text search engines like ElasticSearch, and time series databases like InfluxDB. They are challenging to build because they run at scale and need complex distributed functionality like data replication, fault tolerance, and update consistency. DPA makes building these systems easier by allowing developers to construct them from purely single-node components while automatically providing these critical properties. In DPA, we view a query serving system as a collection of stateful actors, each encapsulating a partition of data. DPA provides parallel operators that enable consistent, atomic, and fault-tolerant parallel updates and queries over data stored in actors. We have used DPA to build a new query serving system, a simplified data warehouse based on the single-node database MonetDB, and enhance existing ones, such as Druid, Solr, and MongoDB, adding missing user-requested features such as load balancing and elasticity. We show that DPA can distribute a system in <1K lines of code (>10× less than typical implementations in current systems) while achieving state-of-the-art performance and adding rich functionality.

1 Introduction

Specialized systems that perform data-parallel, low-latency computations and frequent bulk data updates are becoming ubiquitous. These *query serving systems* include search engines like ElasticSearch and Solr [9, 13], online analytics (OLAP) systems like Druid and Clickhouse [11, 70], time-series databases like InfluxDB and OpenTSDB [14, 17], and many others [10, 15, 18, 21, 40, 51, 52]. These systems are critical to everyday applications: for example, Walmart uses ElasticSearch to check purchases for fraud in real time [6], Target and Capital One use Druid and InfluxDB for real-time monitoring in their production services [5, 7], and Facebook developed Unicorn [40] to provide graph-based search.

Developing query serving systems is challenging because their workloads typically run at large scale. Therefore, query serving system developers must implement complex distributed functionality, including data replication, update consistency, fault tolerance, and load balancing. These features vary little between query serving systems, but must be re-implemented in each of them, typically in custom distribution

layers comprising tens of thousands of lines of complex code (e.g., ~70K lines in Druid) written over many person-years. As a result of this complexity, not only are new query serving systems hard to build, but existing ones are difficult to adapt to changing user demands. For example, most query serving systems were designed for fixed-size on-premise clusters, although users increasingly deploy them in the cloud. Therefore, they do not provide user-requested cloud features such as elastic cluster auto-scaling [3, 4]. Adding any new distributed feature to an existing, large codebase can take years, even when there is strong user demand [60, 74].

Ideally, developers would be able to write query serving systems using a high-level programming model that simplifies distributing their data and computations across a cluster. Unfortunately, current distributed programming models do not support the unique workloads of query serving systems, with their combination of data-parallel low-latency queries and frequent bulk data updates. Actor models like Erlang [29], Orleans [35] and Ray [61] can manage mutable state, but lack abstractions, such as consistency and atomicity, for data-parallel operations. Parallel processing frameworks like Spark [72] can execute data-parallel queries, but lack abstractions for managing data, assuming it to be immutable.

In this paper, we propose a new programming model called *data-parallel actors (DPA)* that extends the actor model to support the unique needs of query serving systems. DPA allows developers to construct a distributed query serving system from purely single-node components, as we show in Figure 1. The DPA runtime then automatically provides the system with complex distributed features such as fault tolerance, consistency, load balancing, and elasticity.

Designing a programming model for query serving systems is challenging because of their wildly different query and data models, from search engines to timeseries databases to document stores. DPA's insight is that the distributed functionality of a query serving system can be implemented largely independently of how the system stores and processes data on individual nodes. Therefore, DPA represents a query serving system as a collection of black-box data partitions, each encapsulated in a stateful actor. However, while conventional actor models focus on *concurrency*, where there are many actors but clients only communicate with one at a time, query serving systems also require *parallelism*: one operation can run over data in many actors, often with consistency and atomicity requirements. Thus, DPA provides parallel operators and updates over its stateful actors. Parallel operators let develop-
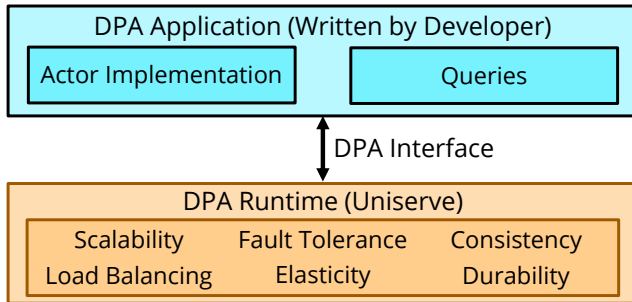
Figure 1: With DPA, a developer can construct a distributed query serving system from single-node components: code for actors and queries (blue) that implement per-node data structures and query processing logic. A DPA runtime like Uniserve (orange) manages actors and executes queries, automatically providing distributed features.

ers construct queries from generic operations such as map and broadcast, while parallel updates offer configurable consistency and atomicity guarantees. DPA defines these operations and enforces their guarantees, but is agnostic to how each node processes its part of the work. Thus, DPA *separates responsibilities* in building a query serving system, so that developers only implement single-node data structures and operations but receive a robust, performant distributed system.

We show that DPA can express the functionality of a wide range of current query serving systems, while adding powerful user-requested features that production systems lack. For example, we used DPA to wrap the existing single-node components in an OLAP system (Druid), search engine (Solr) and NoSQL database (MongoDB) into stateful actors in a few hundred lines of code. The DPA ports match the original systems on standard performance benchmarks, but also automatically receive user-demanded missing features like load balancing and elasticity, improving performance on skewed workloads by up to 3×. DPA's generality makes it a powerful abstraction for developing new query serving systems.

We implement the DPA programming model in a runtime called *Uniserve*. Uniserve manages stateful actors and executes queries, automatically providing distributed features like durability, fault tolerance, load balancing, and elasticity. Because workloads require different implementations of these features, Uniserve allows developers to configure systems' consistency guarantees and load balancing and auto-scaling behavior without modifying their core application code.

To evaluate DPA and Uniserve, we use them to distribute four systems: the three ports discussed above (Druid, MongoDB, and Solr) and a new simplified data warehouse we built based on the single-node database MonetDB [50]. We distribute each system with <1K lines of code. Nevertheless, on standard benchmarks, our ports match the originals' performance, while our data warehouse matches Amazon Redshift and outperforms Spark SQL. Each DPA-based system automatically receives powerful features, including fault tolerance, durability, consistency, load balancing, and elasticity. Some

of these features, particularly load balancing and elasticity, are missing and frequently requested by users in Druid, MongoDB, and Solr. By adding these features, DPA improves these systems' performance by up to 3× on skewed workloads. In summary, our contributions are:

- We identify *query serving systems* as an important emerging class of distributed systems defined by low-latency data-parallel queries and frequent bulk updates. We show that their workloads are not supported by existing high-level distributed programming models.

- We propose *data-parallel actors (DPA)*, a novel programming model for building distributed query serving systems from purely single-node components. We build a DPA runtime, *Uniserve*, which automatically provides fault tolerance, consistency, durability, load balancing, and elasticity to query serving systems built with DPA.

- We demonstrate the power and practicality of DPA by using it to build a simplified data warehouse and porting the popular systems Solr, Druid, and MongoDB to it. Our implementations require <1K lines of code (replacing tens of thousands) but match or outperform current systems while providing rich missing functionality.

2 Background and Motivation

In this section, we give three examples of widely used query serving systems, then make the case for DPA.

2.1 Case Studies

Apache Solr. Solr [9] is a distributed full-text search system. It provides a rich query language for searching text documents and is optimized to serve thousands of queries per second at millisecond latencies. Solr stores documents in inverted indexes based on Apache Lucene [34].

Apache Druid. Druid [70] is a high-performance analytics system. It provides fast ingestion and real-time search and aggregation of time-ordered tabular data, such as machine logs. Druid achieves its high performance through specialized *segment* data structures that store data in a tabular format optimized with summarization, compression, and custom indexes.

MongoDB. MongoDB [15] is a NoSQL document database. Unlike Solr and Druid, it is not primarily an analytics system, but is often used for analytics [16]. MongoDB performs search and aggregation queries over semi-structured data. It uses a schemaless document-oriented data format, backed up by indexes, to give users flexibility in how their data is stored and queried without sacrificing performance.

2.2 Motivating DPA

Solr, Druid, and MongoDB are popular [12] query serving systems that serve different workloads. However, while their physical data structures and query execution strategies are diverse, all use custom distribution layers to distribute data and

	Fault Tolerance	Load Balancing	Elasticity
Solr	✓	X	✓
Druid	✓	X	X
MongoDB	✓	X	X
Uniserve	✓	✓	✓

Table 1: Distributed features of query serving systems.

queries while handling failure and ensuring data consistency. These distribution layers are difficult to implement, requiring tens of thousands of lines of complex code (~90K LoC in Solr, ~70K LoC in Druid, and ~120K LoC in MongoDB).

The difficulty of distributing query serving systems complicates developing new systems, but also causes existing systems to lack user-demanded features. For example, Solr, Druid, and MongoDB struggle to provide load balancing and elasticity, as shown in Table 1. As a result, their users must over-provision clusters [45], go through the difficult and error-prone [3, 4] process of manually integrating external auto-scalers, or risk poor performance when load skews or spikes.

One reason popular systems are missing important features is that the requirements for distributed systems change over time. For example, elasticity is considered important today because most query serving systems run in the cloud, where scaling the size of a cluster is easy. However, many existing systems (including Druid, Solr, and MongoDB) were built when the cloud was less popular, so support for auto-scaling was less important and was not included. Unfortunately, the complexity of query serving systems' distribution layers makes it difficult to add new features when users demand them. For example, adding strongly consistent replication to MongoDB required designing a novel consensus protocol because design choices made early in MongoDB's lifetime precluded using any existing protocol [74]. Similarly, adding support for joins to Druid has been a slow, multi-year process because the system was originally built assuming queries would not require communication between data sources [60].

DPA helps solve these problems by *separating responsibilities* in a query serving system. A developer using DPA is only responsible for the core, unique functionality of their system: storing and querying data. DPA and its runtime Uniserve take responsibility for distribution and scalability, automatically providing distributed features like fault tolerance, consistency, load balancing, and elasticity. As user demands change, new features can be added to Uniserve with minimal modifications to underlying systems. This makes it easier to build new query serving systems and maintain existing ones, as they can obtain state-of-the-art distributed functionality by simply implementing the DPA interface in a ~1K LoC shim layer.

3 DPA Overview and Interface

DPA lets developers construct a distributed query serving system from purely single-node components. To use DPA, a developer must first implement an actor object that encapsu-

lates a data partition like a Solr index or Druid segment. They must then implement a query planner that translates incoming user queries to the DPA parallel operators. We show the interface for actors and operators in Figure 2.

3.1 Actors and Data

In DPA, developers express a query serving system as a collection of stateful single-node actors, each encapsulating a partition of data and exposing methods for manipulating and querying it. Query serving systems use a wide variety of data representations, from Solr inverted indexes to Druid table segments, so DPA actors can encapsulate any data structure the developer chooses for storing a collection of records. We sketch the interface for an actor in Figure 2. DPA views an actor's implementation as a black box. Actors are only required to implement four core methods: create, destroy, serialize, and deserialize (an optional fifth method, *snapshot*, is discussed in §4.2), which the DPA runtime uses for data management. The DPA runtime also maps multiple actors to each physical machine and performs load balancing and auto-scaling. Actors typically implement other methods, e.g., custom methods for querying a search index, which are invoked by DPA operators or update functions when the runtime schedules those to run against an actor. Unlike in some general-purpose actor runtimes, actors in DPA can only communicate through DPA's APIs; they cannot pass arbitrary messages to each other.

DPA actors are organized into *tables*, logical collections of data comprising multiple actors. Tables enable systems to manage multiple datasets and address queries and updates. To partition data across actors in a table, DPA maps records inserted in the system to actors based on *partition keys*; all records with the same key are assigned to the same actor.

3.2 Data Updates

In a conventional actor model, clients communicate with one actor at a time, updating its state directly. In a query serving system, however, users often need to update data partitioned across several actors, typically with consistency or atomicity concerns. Therefore, DPA lets developers implement parallel *update functions*, which update multiple actors in a single table. To perform updates, users implement an UpdateFunction interface with several methods, as shown in Figure 2.

Users invoke update functions on DPA tables and supply them with sets of records to add or change. For example, if a user is maintaining a library catalog in Solr, they might supply an update function with records containing information on new books. The runtime maps the records to actors by partition key, then runs the user's update function on each actor with its corresponding records.

To support the diverse data models of query serving systems, DPA provides configurable consistency and atomicity guarantees for updates. These change how updates are implemented. If developers only require eventually consistent updates, they need only implement an "update" method ap-

Actor Interface

create() → Actor	Create a new (empty) actor.
destroy()	Destroy an actor.
serialize() →File	Serialize an actor's data to files on disk.
deserialize(File) →Actor	Reconstruct an actor from files on disk.
snapshot() →Actor	Create a snapshot of an actor's data.

Record Interface

getPartitionKey() →Int	Get a record's partition key.

Update Function Interface

updatedTableName() →Table	Name of the table to be updated.
consistencyLevel() →Level	What consistency level to use? (§4.2)
update(Actor, List[Record])	Apply eventually consistent update to an actor.
prepare(Actor, List[Record]) → Bool	Prepare a serializable update.
commit(Actor)	Atomically make prepared changes visible.
abort(Actor)	Roll back prepared changes.

Parallel Operator Interface

inputs() →List[Operator|Table]	What are the input operators and tables?
keysToQuery() →Map[Int, List[Int]]	For inputs, what partition keys are used?
operator() → OperatorFunction	The operator function. Signature depends on the operator (see below).

Parallel Operator Functions

map(Actor) → Data	Apply a transformation to data.
scatter(Actor) → List[(K, C)]	Partition data into (attribute, chunk) pairs.
gather(K, List[C], Actor) → Data	Combine chunks with the same attribute, plus actors whose partition key matches that attribute; materialize the output.
query(Actor) → V	Query an actor to obtain a value.
combine(List[V]) → V'	Combine values into a query answer.

Figure 2: The DPA interface. It consists of callback functions implemented by the developer and invoked by Uniserve to manage data and execute queries.

plying an update to an actor. However, if they need stronger guarantees such as serializability, they must implement the participant protocol of two-phase commit (prepare, commit and abort). We discuss consistency in Section 4.2.

3.3 Queries

Unlike traditional actor models, query serving systems execute parallel queries over data stored in many actors. To enable these queries, DPA provides a small but general set of *parallel operators* which let developers construct queries from generic operations like map and broadcast. We list the parallel operators in Figure 2 and diagram them in Figure 3.

Users write queries by subclassing one of several parallel operator classes (e.g., MapOperator) and implementing appropriate callback functions. Queries may be composed of multiple operators. In practice, we expect developers to implement a query planner in their system's client library that translates queries to DPA operators for execution by the DPA runtime. Most existing query serving systems have similar planners. Both the query planning logic and operator execution callbacks can be single-node: the DPA runtime handles the work of distributing a plan's computation by executing each operator on each actor that contains relevant data.

Operators cannot modify actor state (only updates can), but instead materialize output data that later operators can read. The input to each operator is a list of tables and of data materialized by other operators. Operators can specify what partitions of their input data to query through their *keysToQuery* method, listing specific partition keys for each input.

DPA provides five generic parallel operators that we found sufficient to support the serving systems we considered (Sec-

a) Map Operators apply a transformation to several actors in parallel, materializing the transformed data.
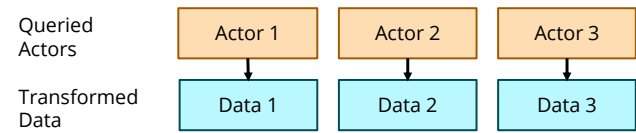
b) Retrieve and Combine Operators compute the result of a query. A retrieve operation computes values from actors in parallel, then a combine operation combines them into a query result.
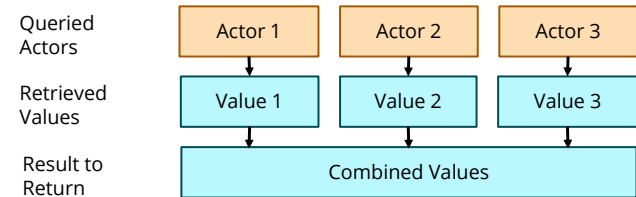
c) Scatter and Gather Operators enable collective operations. A gather operation computes (attribute, data chunk) pairs from actors. A scatter operation combines chunks with the same attribute and materializes the result. Scatter can also (not shown) combine chunks with other actors whose partition key matches the chunk attribute.
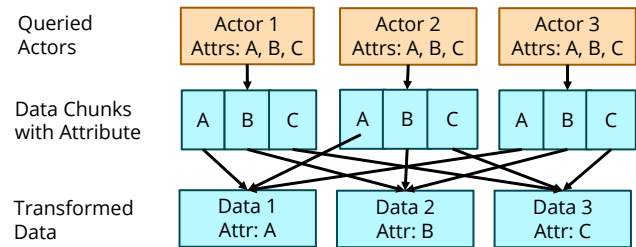

Figure 3: The five DPA parallel operators.

tion 5), though more operators could be added:

Map. The *map* operator applies a function to actors in parallel and materializes the transformed data. For example, a map operator might search for documents in a collection based on a field, or in a subset of actors specified via *keysToQuery*.

Retrieve and Combine. The *retrieve* operator computes a value from an actor and returns it to the DPA client. It is used to retrieve the results of a query. If retrieve is executed on many actors in parallel, it must be followed by a *combine* operator, which aggregates retrieved values. Retrieve and combine must be the last two operators executed in a query. For example, if in Solr we have several actors storing indexed text data and wish to search it for the word "computer," we can execute a retrieve operation to find documents containing the word "computer" on each actor, then combine these results.

Scatter and Gather. The last two operators, *scatter* and *gather*, provide data communication between actors, enabling collective operations such as broadcast and shuffle. The scatter operator produces from an actor a set of (attribute, chunk) pairs, where the attribute can be any value and the chunk contains data stored in a developer-defined serialized format. A

scatter operator must be followed by a gather operator. Gather executes one time for each attribute produced by the preceding scatter. Each execution of gather takes in all data chunks associated with that attribute, along with any actor containing data whose partition key matches the gather attribute, and materializes combined and transformed data.

To demonstrate scatter and gather, consider a shuffle join in a data warehouse setting. Say we have tables of customer data $C(c\_id, country)$ and order data $O(o\_id, c\_id, price)$, both partitioned across several actors. We wish to compute the total amount of money spent by each French customer: SELECT c_id, SUM(price) FROM C, O WHERE C.country='France' GROUP BY c_id. First, we execute a scatter operation on every actor containing data from C or O. This operator returns (attribute, chunk) pairs where every attribute corresponds to a set of customers (range of values of c_id) and every chunk contains data associated with those customers. We then execute a gather operator on the results of the scatter. Each gather execution takes in a unique attribute and all its associated chunks. In other words, it takes in all records from both tables corresponding to a set of customers. The operator executes the original query on this data, computing the amount of money spent by each French customer. Each execution materializes new data containing results for a different set of customers; this data collectively forms the result of the original query. Subsequent operators could then query this data; for example retrieve and combine operators could be used to find the ten top-spending French customers.

3.4 Case Study: Solr

We now describe how to create a DPA port of the distributed full-text search system Solr [9]. Natively, Solr stores data by sharding text documents across Lucene inverted indexes (custom data structures enabling ultra-fast search [34]) on several machines. When Solr receives a new document, it hashes it, uses the hash to pick a shard, and adds it to that shard's index. To port Solr's distributed data storage capabilities to DPA, we encapsulate inverted indexes in actors. We add data to actors in units of Solr documents, which act as DPA records. Just like Solr, we hash documents to obtain a partition key, then use it to assign them to actors.

All Solr queries are searches: they take in a criterion, such as a query string, and return a list of documents that satisfy it. This list may be aggregated by grouping or faceting. Natively, Solr distributes queries by searching each shard separately, then combining results on a single node [20]. To port Solr's distributed query capabilities to DPA, we must translate Solr queries to DPA queries. Because all Solr queries are searches, we can implement them using DPA retrieve and combine operators. Each retrieve operator searches its target actor for a set of results, then the results are combined and returned. For example, in a query that searches for books whose title contains the word "goblin," retrieve operators run in parallel on every queried actor, searching their data for "goblin." A
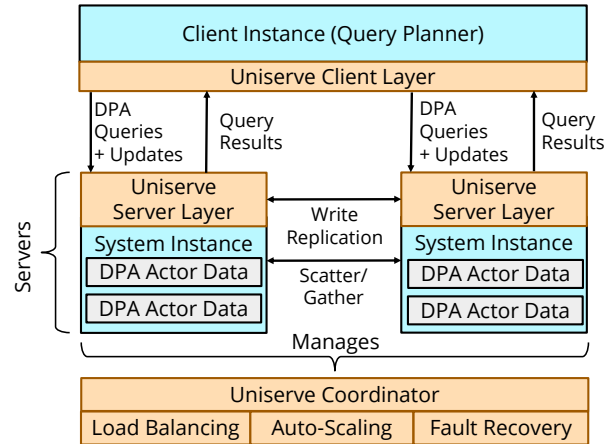

Figure 4: The Uniserve architecture. Servers run a thin Uniserve layer (orange) above actors encapsulating partitions of data (gray) stored in instances of the underlying system (blue). A coordinator manages cluster state and provides distributed features.

combine operator then combines the results. The DPA port of Solr is implemented in <1K lines of code (replacing ~90K lines of native Solr code), and can execute any query recognized by the standard Solr parser. As we show in Section 7, our port matches native Solr performance while providing features lacking in native Solr, such as load balancing.

4 Uniserve: A Runtime for DPA

We implement DPA in a runtime called Uniserve. In the DPA programming model, developers take responsibility for implementing actors and queries on a single node, but Uniserve takes responsibility for distributing them, managing actors and executing queries at scale. Uniserve automatically provides critical distributed features such as fault tolerance, durability, consistency, load balancing, and elasticity.

4.1 Architecture

A Uniserve cluster consists of many data servers. Each runs a thin Uniserve layer over developer-provided single-node code responsible for physical data storage. Clients send queries and updates to servers. Each client runs a thin Uniserve layer above a developer-provided query planner. A central coordinator manages cluster state with the help of ZooKeeper [49]. Uniserve additionally requires an external durable storage system (e.g. S3 or HDFS) to back up data. We diagram the Uniserve cluster architecture in Figure 4.

Servers store data and execute queries. In each server, a thin Uniserve layer runs above developer-provided single-node code responsible for physical data storage and query execution. For example, if we were to distribute Solr using DPA and Uniserve, each server would run a Uniserve layer above a single-node Solr instance. DPA actors encapsulate physical partitions of data stored in this system, so for example each actor might encapsulate a Solr inverted index. The Uniserve layer facilitates query execution. It receives query operators

and updates from clients and executes them in the underlying system using the DPA interface. Additionally, it handles update replication, maintains a log of the most recent updates, periodically backs up data to durable storage, and transfers actors between servers in response to coordinator commands.

Clients plan queries and submit them to servers. In each client, a thin Uniserve layer runs alongside a developer-provided query planner. The query planner receives user queries (or update requests) in some query language and translates them to DPA parallel operators (or update functions). The client then submits these to the appropriate servers, eventually receiving and returning a result. Clients learn actor locations from the coordinator and ZooKeeper so they know to which servers to send queries or updates.

A Uniserve cluster contains a single coordinator that manages cluster state. It is responsible for many distributed capabilities including load balancing, failure recovery, and elasticity, which we discuss in more detail later. It backs up cluster state to ZooKeeper. To minimize query latency at scale, the coordinator is entirely off the query critical path.

4.2 Update Consistency and Atomicity

Conventional actor runtimes do not provide cross-actor data consistency guarantees, assuming operations occur on a single actor at a time. Query serving systems, however, perform parallel updates on partitioned and replicated data, so Uniserve provides cross-actor consistency and atomicity guarantees.

Query serving systems typically ingest bulk data for analytics; for example time series in Druid or logs in Solr. Updates are usually append-only, but modification of existing data is possible. Most updates are batched, and systems provide high update throughput but not necessarily low update latency. Many systems, like Druid, do *not* support transactional semantics. However, they still provide update consistency and atomicity guarantees of varying strength.

Uniserve automatically provides primary-backup actor replication and data consistency guarantees to query serving systems. Because query serving system data models vary, we make these guarantees *configurable*: when implementing an update function (§3.2), developers can choose a level of consistency appropriate to their data model. Uniserve provides the consistency levels most common in the query serving systems we surveyed. In the remainder of this section, we describe these guarantees and what developers must implement to obtain them. Then, in Section 4.3, we explain how Uniserve upholds its guarantees in case of failures.

Eventual Consistency. By default, Uniserve provides eventual consistency, guaranteeing only that all replicas of an actor eventually converge to the same state. Many systems, like Solr and Druid, use eventual consistency [19]. To write an eventually consistent update function, developers need only implement an "update" method. To execute an eventually consistent update, Uniserve applies it to the primary of an actor synchronously, then replicates it asynchronously. All replicas

of an actor apply the same updates in the same order.

Serializable Updates. Uniserve can guarantee serializability for updates, so the outcome of a sequence of updates is equivalent to the outcome of the updates executed serially. As implemented, this also guarantees linearizability, so read queries made after an update completes always reflect the update. This functionality was recently added to MongoDB [74] and is common in data warehouses. To write a serializable update function, developers must implement the participant protocol of two-phase commit, with separate prepare, commit, and abort stages. Uniserve only commits an update if it has successfully prepared on all actors and their replicas, aborting if failures occur. We currently do not allow multiple serializable updates to run concurrently on the same table, but plan to add concurrency control in the future.

Full Serializability. Uniserve can make updates serializable (and therefore atomic) with respect to read queries, so a parallel read query either sees an update applied to all actors or to none of them, as in SQL databases. To obtain this guarantee, developers must both provide serializable update functions and implement the optional *snapshot* actor method (Figure 2). Using this method, Uniserve creates a versioned copy of each actor's data upon update and ensures that read queries see consistent data versions across actors. We expect developers to implement snapshot using optimizations such as shadow paging and copy-on-write to minimize its cost.

4.3 Fault Tolerance and Failure Recovery

Uniserve assumes a fail-stop model for failures, where the only way servers fail is by crashing. It also assumes that if a server crashes, it remains crashed until restarting (when it will be treated as a new server). Moreover, it assumes the coordinator and ZooKeeper are always available; if either fails the cluster will be unavailable until they are restarted, with the coordinator restoring its state from ZooKeeper.

Durability. Uniserve provides update durability through replication and through asynchronous backup to durable storage such as S3. If all replicas of an actor fail, the coordinator orders a random surviving server to load the actor from durable storage. Thus, Uniserve can only lose data if all replicas of an actor fail, and will only lose data committed since the last backup. Additionally, eventually consistent updates can be lost if the primary fails before the updates are replicated.

Update Fault Tolerance. When providing eventual consistency, Uniserve only guarantees that all replicas of an actor will eventually converge to the same state. Therefore, it is possible for an update to partially succeed—to succeed on some actors but fail on others. If the primary of an actor fails, the coordinator chooses the replica with the most advanced update as the new primary, relying on the guarantee that all replicas apply the same updates in the same order. All other replicas then sync with the new primary, applying missing updates from its log to converge to its state.

When providing serializability, Uniserve guarantees that all updates either totally succeed or abort. Uniserve only commits an update if it has successfully prepared on all actors and their replicas, aborting if any failures occur. To ensure the cluster remains in a consistent state in case of a client crash, the client writes ahead any commit or abort decision to ZooKeeper; servers can reference this if the client fails (or abort if the client fails before making a decision).

Query Fault Tolerance. If a failure occurs during the execution of a parallel operator on an actor, the client retries with a different replica. It keeps retrying until it has exhausted all replicas; this occurs only if all are lost, in which case the actor must be restored from durable storage and the query fails.

4.4 Load Balancing and Data Placement

Query serving systems often have unpredictable workloads skewed towards a small number of data items or partitions, so load balancing is necessary for consistent performance [45]. The obvious way to balance load is through fine-grained query scheduling, but this is impractical for query serving systems because of their strict latency requirements and because all queries must run on specific data partitions. Instead, Uniserve balances load through data placement, managing the actor-to-server assignment to ensure no server is overloaded.

By default, Uniserve provides a greedy load balancing algorithm, similar to that of E-Store [67], which repeatedly moves the most-loaded actors from the most-loaded servers to the least-loaded servers while also replicating actors whose load exceeds average server load. However, some applications may want to instead use a custom algorithm. Therefore, we allow developers to define a *data placement policy*, which uses information on cluster utilization to compute an assignment of actors to servers. If a policy is provided, Uniserve takes responsibility for collecting its input data and implementing its output assignment, moving actors to new locations.

A data placement policy must be expressed as a function that takes in the total query load (self-reported by the underlying system) and memory and disk usage of each actor, as well as the current assignment of actors to servers. It returns an updated assignment of actors to servers, expressed as a map from actor number to a list of server IDs. Assignments may replicate actors across multiple servers, either for redundancy or to spread out their load.

To physically move actors during load balancing, Uniserve first prefetches, from durable storage, replicas of reassigned actors on their target servers. These then sync with the actor primary, applying updates from its log. Only after replicas are ready does Uniserve notify clients of the actor movement. Then, after notifying clients, it deletes the original copies of the actors if necessary. If some of the deleted actors were primaries, Uniserve designates randomly selected replicas as new primaries. This procedure ensures high query availability during shard transfer, but if a primary is removed updates may briefly block while a new primary is designated.

	System Type	Data Type	Query Operations
Druid [70]	OLAP	Indexed Tables	Aggregations, joins
Pinot [51]	OLAP	Indexed Tables	Aggregations
ClickHouse [11]	OLAP	Indexed Tables	Aggregations, joins
Atlas [10]	Timeseries DB	Time series	Aggregations
InfluxDB [14]	Timeseries DB	Time series	Aggregations
Solr [9]	Full-Text Search	Indexed text	Text search
ElasticSearch [13]	Full-Text Search	Indexed text	Text search
Unicorn [40]	Graph Database	Social Graphs	Graph Search
FAISS [52]	Vector Database	Vectors	Vector Search
Pinecone [18]	Vector Database	Vectors	Vector Search
Vespa [21]	Vector Database	Vectors	Vector Search
MongoDB [15]	NoSQL	Documents	Aggregations, search
MonetDB [50]	Data Warehouse	Relational Tables	SQL

Table 2: Systems we believe can be distributed with or ported to DPA and their properties. Systems we have implemented are in bold.

4.5 Elasticity and Auto-Scaling

Query serving system load often varies over time, so they benefit from *elasticity*, the ability to dynamically adjust cluster size. As a result, when deployed in an elastic cloud environment such as EC2, Uniserve automatically scales cluster size in response to load changes.

By default, Uniserve provides a utilization-based auto-scaling algorithm similar to the algorithms used in cloud auto-scalers [32]. It adds servers if CPU utilization exceeds an upper threshold and removes them if it is below a lower threshold. However, like in load balancing, Uniserve also gives developers the option of defining their own *auto-scaling policy*, which uses information on cluster utilization to decide whether to add or remove nodes. Uniserve provides the policy with its input and physically executes its commands, adding or removing nodes and transferring actors as necessary.

An auto-scaling policy must be defined as a function that takes in the CPU utilization, memory and disk usage, and total query load of each server. It returns the number of servers to be added or removed, as well as the IDs of the servers to be removed, if any (chosen randomly if there is no preference).

Uniserve periodically executes the policy (using a configurable interval) and adjusts cluster size. After adding or removing a server, Uniserve uses the load balancer to reassign actors; if servers are removed this reassignment is done preemptively so availability is not affected.

5 Generality of DPA

In this section, we demonstrate the generality of DPA by describing some of the diverse systems it can distribute, summarized in Table 2. We also discuss its limitations.

OLAP Systems and Time Series Databases. OLAP systems rapidly answer multidimensional analytics queries over tables. They are closely related to time series databases, which query time-ordered data. Both typically store data in a compressed and indexed columnar format. Their workloads usually filter, group, and aggregate this data. This naturally fits DPA: we partition data by key columns across actors (e.g., by time range) to support partition filtering, and implement

most aggregations with retrieve and combine operators, using scatter and gather to shuffle or broadcast data if necessary. We implement a port of Druid [70], which is both an OLAP system and timeseries database, on DPA; its design patterns generalize to others from both categories such as Pinot [51], Clickhouse [11], Atlas [10] and InfluxDB [14].

Full-Text Search. Full-text search systems execute search queries over text data stored in specialized data structures such as inverted indexes [34]. Because all their queries are searches, they are easy to fit to DPA, as we showed in Section 3.4. We implement a port of one full-text search system, Solr [9], and can generalize to others like ElasticSearch [13].

Vector Databases. Vector databases store data using vector indexes to perform fast nearest neighbor search, often for machine learning workloads. Recent examples are Pinecone [18], Vespa [21], and FAISS [52]. Like full-text search systems, they easily fit DPA as their queries are searches.

Graph Databases. Graph databases represent data using a graph data model. Some graph database queries are data-parallel, including whole-graph algorithms like PageRank and queries like finding all checkins at a certain location in a social network graph [40]. Others are not; for example, a graph traversal query, like finding all nodes within N hops of a target, is most efficiently implemented using breadth-first search, not data-parallel operators such as iterative self-joins. Data-parallel graph databases such as Facebook's Unicorn [40] search engine fit the DPA programming model.

Other Systems. DPA can distribute other systems with data-parallel queries. For example, we implement a DPA port of the NoSQL document store MongoDB. We also implement a simplified OLAP data warehouse based on the single-node columnar database MonetDB.

Limitations of DPA. DPA has two major limitations. First, its query model works best for data-parallel queries. As we have shown, this is sufficient for many popular query serving systems, but not some specialized query types like graph traversal queries. Nonetheless, we believe DPA would have made many of today's query serving systems easier to develop, and can augment them with missing functionality.

Second, DPA is not designed to provide low latency for small point updates, especially with transactional guarantees. Small transactional updates are rare in query serving systems because these are often updated in bulk (e.g., using data collected in a message queue like Kafka). However, they are common in other contexts such as online transactional processing (OLTP) workloads, which DPA does not target.

6 Distributing Systems with DPA

To demonstrate the practicality of DPA, we use it to distribute four systems. First, we port Druid, Solr, and MongoDB to DPA, replacing their native distribution layers. Then, we build a new system using DPA: a simplified data warehouse based on the single-node column store MonetDB.

We implement each of our four systems in <1K lines of code (LoC). This number includes all code needed to implement the DPA interfaces with each system's already-existing single-node implementation, but not any code in Uniserve. This demonstrates that DPA simplifies building distributed query serving systems, as it replaces custom distribution layers totaling ~90K LoC in Solr, ~120K LoC in MongoDB, and ~70K LoC in Druid. For comparison, Uniserve itself is ~10K LoC. This smaller size is because Uniserve makes use of tools like ZooKeeper and gRPC for basic functionality that other systems implemented themselves.

Solr. We described the port of Solr in Section 3.4.

Druid. In our port of Druid [70], actors encapsulate single-node Druid datasources. These are analogous to database tables and are backed by Druid segments, which are optimized tabular stores for timeseries data. We implement most actor manipulation and update functionality using the Druid datasource API. Serializing and deserializing data is easy because Druid segments live in portable directories on disk.

All Druid queries aggregate filtered and grouped data from datasources. Our port supports most common Druid queries: simple aggregations (sums, counts, or averages) of filtered and grouped data. It could easily be extended to support any other query by adding support for more aggregation operators. Our Druid queries use retrieve and combine operators to separately query actors then aggregate the results. Druid uses a similar model natively. We can also use scatter and gather operators to support Druid's recently-added [60] broadcast joins.

MongoDB. In our port of MongoDB [15], actors encapsulate single-node MongoDB collections, analogous to database tables. We implement most actor manipulation and update functionality using the MongoDB API for manipulating collections. We implement actor data serialization and deserialization using the `mongodump` and `mongorestore` tools.

MongoDB queries apply an "aggregation pipeline" of operators to a collection. These operators perform tasks such as filtering, grouping, and accumulating documents. We can support any MongoDB operator, but so far have only implemented operations for filtering, projecting, summing, counting, and grouping data. Our query implementations are similar to those in our Druid port and those in native MongoDB: querying actors separately, then combining the results.

MonetDB. We have built using DPA a simplified data warehouse based on the single-node column store MonetDB [50]. It stores data in MonetDBLite [65], the embedded implementation of MonetDB. Each server runs MonetDBLite embedded in the same JVM as the Uniserve layer. Actors encapsulate MonetDB tables and implement interface methods using equivalents in the MonetDBLite API.

Our simplified data warehouse supports a large subset of

SQL, including selection, projection, equijoins, grouping, and aggregation. We implement simple aggregation queries with retrieve and combine operators, as in other systems. To execute more complex queries, such as joins, we use scatter and gather operators to shuffle or broadcast data, then use retrieve and combine operators to produce a query result.

7 Experimental Evaluation

We evaluate DPA and Uniserve using the four systems discussed in Section 6. As we have shown, DPA makes distributing these systems considerably simpler; each requires <1K lines of code to distribute as compared to the tens of thousands of lines in custom distribution layers (~90K in Solr, ~120K in MongoDB, and ~70K in Druid). Our evaluation shows that:

1. Distributed systems built using DPA and a specialized single-node system, such as our MonetDB-based simplified data warehouse, can match or outperform comparable distributed systems such as Spark-SQL and Redshift.

2. DPA ports of distributed systems match the performance of natively distributed systems under ideal conditions, such as static workloads without load skew.

3. DPA ports of distributed systems provide new features such as elasticity and load balancing and so outperform natively distributed systems under less ideal conditions – workloads that change, have load skew, or have failures.

7.1 Experimental Setup

We run most benchmarks on a cluster of m5d.xlarge AWS instances, each with four CPUs, 16 GB of RAM, and an attached SSD. We evaluate using Apache Solr 8.6.1, Apache Druid 0.20.1, MongoDB 4.2.3, and MonetDBLite-Java 2.39. We use four data servers for smaller-scale benchmarks and forty for large-scale benchmarks. In both cases, an additional node is set aside for the coordinator.

When benchmarking Solr, Druid, and MongoDB natively, we place the master (Solr ZooKeeper instance, Druid coordinator, MongoDB config and `mongos` servers) on a machine by itself and a data server (SolrCloud node, Druid historical, MongoDB server) on each other node. We also disable query caching and set the minimum replication factor to 1.

When benchmarking systems with Uniserve, we use the implementations described in Section 6. We place the Uniserve coordinator and a ZooKeeper server on a machine by themselves and data servers on the other nodes.

7.2 Experiment Workloads

We evaluate each system with a representative workload taken when possible from the system's own benchmarks. All of our comparison systems achieve state-of-the-art performance on their benchmarks, so DPA also achieves state-of-the-art performance by matching them. We benchmark Solr with queries from the Lucene nightly benchmarks [59]. We run each query

on a dataset of 1M Wikipedia documents (more for large-scale benchmarks) taken from the nightly benchmarks. We use two representative nightly benchmark queries–an exact query for the number of documents that include the phrase "is also" and a sloppy query for the number of documents that include a phrase within edit distance four of the phrase "of the."

We benchmark Druid with two of the benchmark queries from the Druid paper [54, 70]. These are TPC-H queries modified by the Druid developers to reflect the strengths of Druid; we run each against 6M rows of TPC-H data. The queries we use are `sum_all`, which sums four columns of data; and `parts_details`, which performs a group-and-aggregate.

We benchmark MongoDB using YCSB [39], simulating an analytics workload. Before running the workload, we insert 10M sequential items (10GB of data) into the database. We run a workload of 100% scans, where each scan retrieves one field from each of uniformly between 1000 and 2000 items. We base our YCSB client implementation on the MongoDB YCSB client from the YCSB GitHub repository [22].

We benchmark our data warehouse using representative TPC-H queries (Q1, Q3, and Q10) at scale factors of 5 and 25, requiring 5GB and 25GB of data respectively.

7.3 Benchmarks

Ideal Conditions. We first benchmark our Solr, Druid, and MongoDB ports on a uniform workload where each data item is equally likely to be queried. We run each benchmark with several client workers; each repeatedly makes the query and waits for it to complete, recording throughput and latency. We start with a single worker and add more until throughput no longer increases, showing results in Figure 5. We find that, as expected, our ports' performance is similar to native system performance on all benchmarks.

Scalability. We next evaluate Uniserve scalability, scaling the Solr benchmarks with one client worker from four to forty servers. We scale the amount of data to maintain a constant 5 GB of data per server. We show results in Figure 6. Because all queries access all data, we expect performance to be near-constant as the number of servers (and amount of data) increases, and indeed it is.

Data Warehouse Benchmarks. We next benchmark our simplified data warehouse based on MonetDB, comparing its performance with native MonetDB, Spark-SQL [28], and Redshift [47]. We use three TPC-H queries: Q1, an aggregation query; Q3, a three-way join; and Q10, a four-way join. We implement Q3 and Q10 using scatter and gather operators to perform both broadcast and shuffle joins. We show results in Figure 7. We run multiple trials of each benchmark, reporting the average of results after performance stabilizes. This ensures Spark-SQL and Redshift can cache data in memory.

We first investigate the overhead Uniserve adds to single-node MonetDB. On a single node, our data warehouse performs the same as native MonetDB on the aggregation query
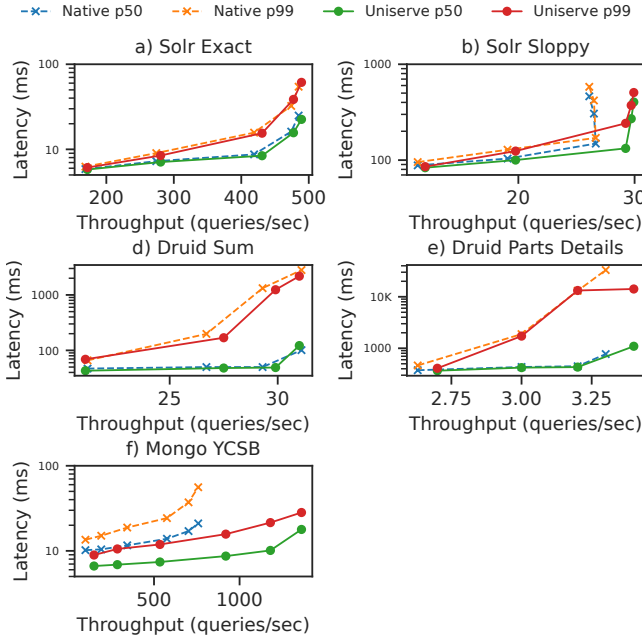
Figure 5: Throughput versus latency for native systems and DPA ports on uniform and static query workloads. Our ports match native system performance.
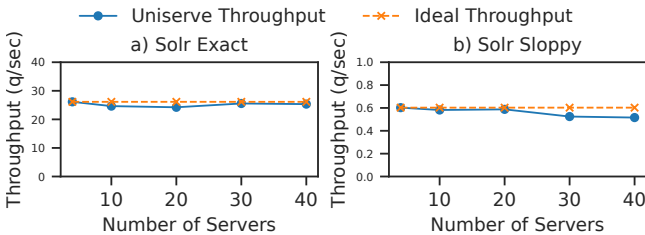

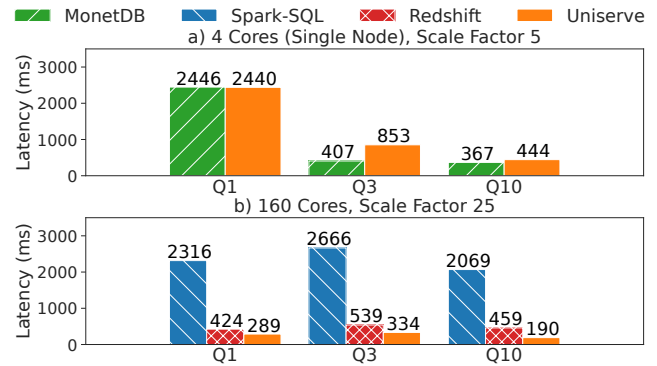**Figure 6:** Uniserve scalability on the Solr benchmarks.

Figure 7: Comparison between our simplified data warehouse, single-node MonetDB, Spark-SQL, and Redshift on TPC-H queries Q1, Q3, and Q10 on 4 cores (single-node) and on 160 cores with TPC-H scale factors of 5 and 25. Uniserve is competitive on a single node and outperforms Spark-SQL and matches Redshift at scale.
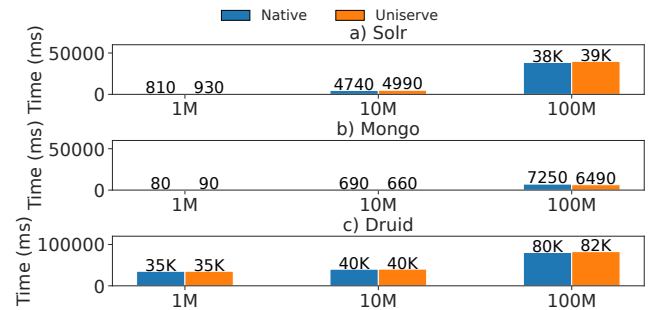
Figure 8: Execution time of 1 MB, 10 MB, and 100 MB updates with native systems and with Uniserve. Uniserve matches native system performance.

Q1 but worse on Q3 and Q10 due to the communication cost of shuffling. We then compare our system to Spark-SQL and Redshift on 160 cores (forty servers for Uniserve and Spark-SQL, five dc2.8xlarge Redshift servers). We find that our data warehouse outperforms Spark-SQL and matches Redshift. This shows that by distributing a single-node system like MonetDB, DPA can in <1K lines of code match or outperform popular distributed systems like Redshift and Spark-SQL on their core workloads.

Update Performance. We next investigate Uniserve update performance. We benchmark 1 MB, 10 MB, and 100 MB updates on Solr, Druid, and MongoDB, using each system's benchmark dataset. We use these bulk writes because they are typical of query serving system workloads. We compare native system performance to Uniserve performance, showing results in Figure 8. For Solr and Druid, we provide eventual consistency, matching those systems' semantics; for MongoDB we enable update serializability (through two-phase commit in Uniserve) and perform the update on four partitions in parallel. We find that across the board, Uniserve matches

native system update performance.

Hotspots. To demonstrate the importance of load balancing, we next investigate the performance of the default Uniserve load balancer on benchmarks with load skew. We compare against Druid, whose load balancer ensures each server hosts the same amount of data but does not balance query load. First, we execute a workload where $7/8$ of the queries are sent to a single slice of data (four months) and scatter the rest uniformly on the remainder of the data, showing results in Figure 9a. Because Uniserve balances load in the hotspot, it outperforms Druid by up to $3\times$.

We next repeat the experiment, fixing the number of clients at twelve but varying the fraction of queries sent to the hotspot. We show results in Figure 9b. We find that changing skew does not affect Uniserve performance because Uniserve keeps load balanced under any load distribution. However, Druid performance worsens with increasing skew.

Dynamic Load. To demonstrate the importance of elasticity in query serving systems, we next investigate the performance of the default Uniserve auto-scaler on a dynamic workload. We run the Solr sloppy benchmark for six hours sending
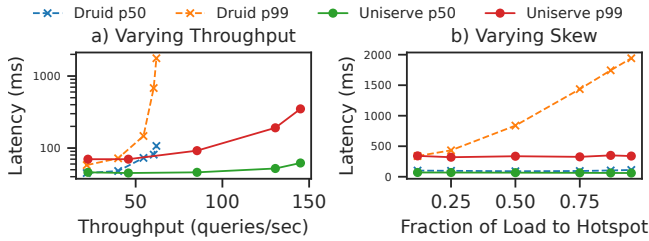
Figure 9: Effect of query skew and load balancing for Druid- and Uniserve-distributed queries. On the left, we vary throughput in a workload where one slice of data receives 7/8 of queries; Uniserve balances load and so outperforms Druid. On the right, we vary the fraction of queries received by the hot slice; Uniserve keeps performance constant as skew increases but Druid does not.

queries at a target throughput, which varies from 240 to 1300 uniformly distributed queries per minute. Uniserve starts with one server and adds or removes more as load changes. We show results in Figure 10. We see that Uniserve is always able to scale to meet the target throughput. As load increases, it adds servers so there are always enough to process each query in time. As load decreases, it removes unnecessary servers but keeps enough to process incoming queries. Because the target query runs in parallel on all actors, adding servers decreases latency (as the query can run in parallel on more cores on more servers) and removing servers increases latency.

Importantly, Uniserve can resize clusters without losing performance. By prefetching replicas of moved actors onto new servers before serving any queries, Uniserve guarantees that queries need not contend with actor transfers for resources. As a result, Uniserve can add or remove servers without affecting throughput or median latency. Tail latency does spike briefly when a server is added, but this represents only the handful of queries sent between when Uniserve notifies servers of the new server and when it notifies clients.

Failures. We next investigate how Uniserve deals with server failures, using the Druid `sum_all` benchmark. We run this benchmark for ten minutes with a client sending 500 asynchronous queries uniformly per minute. Three minutes into the benchmark, we `kill -9` a data server. We record how many queries succeed during each minute of the benchmark. We run the benchmark twice, once starting with four replicas of each data partition or actor (one on each server), and once with just a single replica. We show results in Figure 11.

When all servers have replicas of all partitions (11b), Uniserve recovers instantly, routing queries to replicas. Druid, however, takes thirty seconds to route queries to replicas, resulting in hundreds of query failures. When there is only one replica of each partition (11a), both systems fail hundreds of queries but recover within thirty seconds by restoring replicas from durable cloud storage. However, while all queries sent to Uniserve either fail or successfully complete, some "successful" Druid queries return incorrect results. This experiment confirms previously-reported issues Druid faces in large-scale
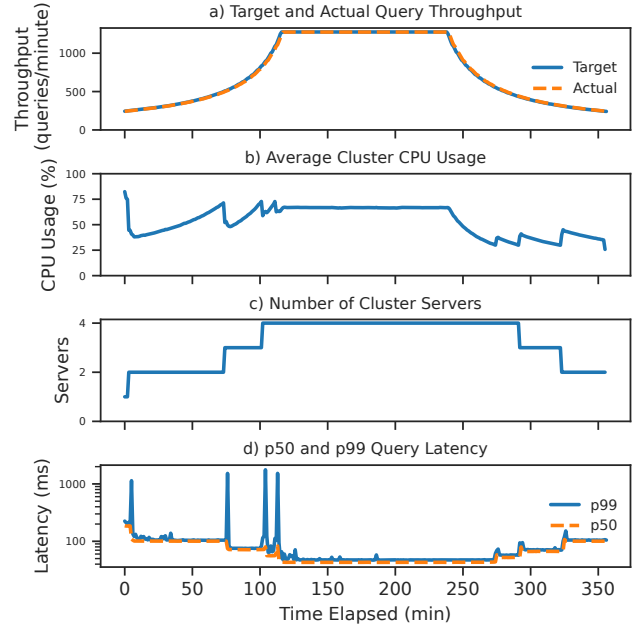

Figure 10: On the Solr sloppy benchmark with Uniserve auto-scaling, varying target throughput and observing effects on actual throughput, average cluster CPU usage, the number of cluster servers, and query latencies. Uniserve scales the cluster so that actual throughput always matches target throughput; resizing causes only brief (<1 sec) spikes in query latency. Latency decreases as cluster size increases because all queries run on all data and their parallelism increases as the data is spread over more servers.

deployments [56] and shows Uniserve can address them.

8 Related Work

Actors Actor models are abstractions for concurrent computation built around stateful agents called actors [26]. Prior surveys [53] identified five characteristics of an actor model: actors encapsulate their own state, exhibit location transparency, are mobile, are scheduled fairly, and communicate through message passing. DPA actors exhibit four of these properties: they encapsulate shards of data, are addressable through partition keys, can be moved between servers, and share resources on each machine. However, unlike prior actor models, DPA actors do not communicate via message passing but are instead acted on by parallel operators and updates. Other systems based on actors include Erlang [30], a programming langauge with built-in actor support; Akka [8], which supports actors on the JVM, including persistent actors with durable state; Orleans [33, 35], which supports virtual actors that are only instantiated on-demand when required; and Ray [61, 69], where developers can call remote procedures on stateful actors.

Critically, most existing actor models focus on *concurrent* computations, not the *parallel* ones performed by query serving systems and DPA. Most actor models do not support cross-actor transactions, requiring users to manually implement protocols such as two-phase commit. Even in systems
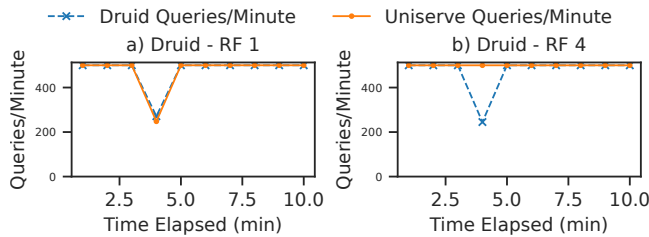
Figure 11: Query throughput (targeting 500 queries/min) of Druid- and Uniserve-distributed `sum_all` queries when one data server is killed after three minutes. The left graph shows performance starting with a single replica of each partition; the right graph with four.

like Ray which allow many actors to be accessed in parallel, there is only limited support for collective operators like gather or scatter [23] and no support for consistency or atomicity guarantees across actors. DPA instead reasons about parallel operators directly, taking advantage of the fact that query serving systems mostly need to support bulk updates as opposed to many concurrent write transactions, and offers multiple consistency levels to support different system designs. Eldeeb and Bernstein extended Orleans with a transactional actor concept [43], but that work focused on allowing clients to make multiple calls to the same actor as a single transaction and tracking these calls' effects on downstream actors through message passing, which would be expensive for the large data-parallel operations that DPA targets. Early versions of Akka also supported transactional actors on the same server [2], but this was removed because the mechanism was hard to extend to multiple servers [1].

Other Distributed Programming Models   One class of programming model often used for parallel queries are batch frameworks like MapReduce [42], Hadoop [66], Percolator [64], Dryad [71], and Spark [72]. Unlike query serving systems, these only execute computations and do not provide abstractions for managing data, typically assuming its immutability. Moreover, they are not designed for low latency and typically do not implement many of the optimizations used in query serving systems, such as augmenting data with secondary indexes. Researchers have attempted to build updatable data structures over Spark RDDs, such as PART [41], but these are greatly limited by the immutability of RDDs.

Streaming and dataflow systems like Spark Streaming [27, 73], Naiad [62], and Flink [36] execute queries in real time on streaming data. However, unlike query serving systems, they focus primarily on continuous computation (incrementally updating the result of a query as data comes in) and do not perform data management or low-latency query serving. They are often used to write data into a query serving system.

Cluster management systems like Helix [46], Mesos [48], and YARN [68] are designed to deploy distributed systems at scale. Mesos and YARN are primarily concerned with assigning resources to each application. Helix, like Uniserve, automatically manages the applications running on it, providing features such as elasticity and fault tolerance. However, it is not designed for query serving workloads and lacks a query model and abstractions for consistency and atomicity.

Auto-sharding systems like Slicer [25], Centrifuge [24], and Shard Manager [55] assign data and queries to shards based on partition keys, like DPA. Slicer and Centrifuge *only* manage key affinity, telling applications what keys are assigned to what servers. Shard Manager goes further and manages data placement, moving data shards between servers. However, unlike Uniserve, these systems do not provide high-level abstractions on top of shards, such as parallel operators and updates with consistency and atomicity guarantees.

Thor [58] stores data in persistent distributed objects for heterogeneous applications to access. These objects resemble DPA actors, but Thor must run object operations on client machines and does not provide high-level abstractions such as a query model or configurable consistency guarantees.

Middleware systems for databases automatically distribute data and queries across existing database installations and provide features like fault tolerance [57, 63] and load balancing [31]. However, these solutions are typically specialized to particular database types, like relational databases [37, 38] or NoSQL stores [44], and do not provide general abstractions to support a wide range of data and query models like DPA.

9   Conclusion

Query serving systems are an important emerging class of distributed systems that power many Internet applications. Traditionally, they are implemented from scratch, requiring substantial effort to add distributed query processing and data management functionality. We presented *data-parallel actors (DPA)*, a high-level programming model that allows developers to build reliable and performant distributed query serving systems from single-node data structures and logic. We showed that DPA can express the functionality of a wide range of query serving systems by building a simplified data warehouse and porting Druid, Solr, and MongoDB to DPA in <1K lines of code, matching performance and adding rich missing functionality such as automatic load balancing and auto-scaling. We believe DPA is a valuable tool to help organizations more easily develop these important systems.

References

[1] Akka 2.4 migration guide. https://doc.akka.io/docs/akka/2.4/project/migration-guide-2.3.x-2.4.x.html.

[2] Akka transactors documentation. https://doc.akka.io/docs/akka/2.2/scala/transactors.html, 2015.

[3] How to Setup ElasticSearch Cluster with Auto-Scaling on Amazon EC2? https://stackoverflow.com/questions/18010752/, 2015.

[4] MongoDB Cluster with AWS Cloud Formation and Auto-Scaling. https://stackoverflow.com/questions/30790038/, 2016.

[5] Why Architecting for Disaster Recovery is Important for Your Time Series Data. https://www.influxdata.com/customer/capital-one/, 2018.

[6] How Walmart is Combating Fraud and Saving Consumers Millions. https://www.elastic.co/elasticon/tour/2019/dallas/, 2019.

[7] Enterprise Scale Analytics Platform Powered by Druid at Target. https://imply.io/virtual-druid-summit, 2020.

[8] Akka. https://akka.io/, 2021.

[9] Apache Solr. https://lucene.apache.org/solr/, 2021.

[10] Atlas. https://github.com/Netflix/atlas, 2021.

[11] ClickHouse. https://clickhouse.tech/, 2021.

[12] DB-Engines Ranking. https://db-engines.com/en/ranking, 2021.

[13] Elasticsearch. www.elastic.co, 2021.

[14] InfluxDB. https://www.influxdata.com/, 2021.

[15] MongoDB. https://www.mongodb.com/, 2021.

[16] MongoDB for Analytics. https://www.mongodb.com/analytics, 2021.

[17] OpenTSDB. http://opentsdb.net/, 2021.

[18] Pinecone. https://www.pinecone.io/, 2021.

[19] Shards and Indexing Data in SolrCloud, Aug 2021.

[20] Solr Distributed Requests. https://solr.apache.org/guide/8_8/distributed-requests.html, 2021.

[21] Vespa. https://vespa.ai/, 2021.

[22] YCSB GitHub. https://github.com/brianfrankcooper/YCSB, 2021.

[23] Ray collective communication. https://docs.ray.io/en/latest/ray-collective.html, 2022.

[24] Atul Adya, John Dunagan, and Alec Wolman. Centrifuge: Integrated lease management and partitioning for cloud services. In *NSDI*, volume 10, pages 1–16, 2010.

[25] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, et al. Slicer: Auto-sharding for Datacenter Applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 739–753, 2016.

[26] Gul A Agha. Actors: A model of concurrent computation in distributed systems. Technical report, Massachusetts Inst of Tech Cambridge Artificial Intelligence Lab, 1985.

[27] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. Structured streaming: A declarative api for real-time applications in apache spark. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 601–613, New York, NY, USA, 2018. Association for Computing Machinery.

[28] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394, 2015.

[29] Joe Armstrong. A History of Erlang. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, pages 6–1, 2007.

[30] Joe Armstrong. Erlang. *Communications of the ACM*, 53(9):68–75, September 2010.

[31] Jaiganesh Balasubramanian, Douglas C Schmidt, Lawrence Dowdy, and Ossama Othman. Evaluating the Performance of Middleware Load Balancing Strategies. In *Proceedings. Eighth IEEE International Enterprise Distributed Object Computing Conference, 2004. EDOC 2004.*, pages 135–146. IEEE, 2004.

[32] Jeff Barr. New AWS Auto Scaling – Unified Scaling For Your Cloud Applications. 2018.

[33] Phil Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. Orleans: Distributed virtual actors for programmability and scalability. Technical Report MSR-TR-2014-41, March 2014.

[34] Andrzej Białecki, Robert Muir, and Grant Ingersoll. Apache Lucene 4. In *SIGIR 2012 Workshop on Open Source Information Retrieval*, page 17, 2012.

[35] Sergey Bykov, Alan Geller, Gabriel Kliot, James R Larus, Ravi Pandya, and Jorgen Thelin. Orleans: Cloud Computing for Everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, pages 1–14, 2011.

[36] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[37] Emmanuel Cecchet, George Candea, and Anastasia Ailamaki. Middleware-Based Database Replication: the Gaps Between Theory and Practice. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 739–752, 2008.

[38] Emmanuel Cecchet, Marguerite Julie, and Willy Zwaenepoel. C-JDBC: Flexible Database Clustering Middleware. In *USENIX Annual Technical Conference*, number CONF, 2004.

[39] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[40] Michael Curtiss, Iain Becker, Tudor Bosman, Sergey Doroshenko, Lucian Grijincu, Tom Jackson, Sandhya Kunnatur, Soren Lassen, Philip Pronin, Sriram Sankar, et al. Unicorn: A system for searching the social graph. *Proceedings of the VLDB Endowment*, 6(11):1150–1161, 2013.

[41] Ankur Dave, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. Persistent adaptive radix trees: Efficient fine-grained updates to immutable data.

[42] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. 2004.

[43] Tamer Eldeeb and Phil Bernstein. Transactions for distributed actors in the cloud. Technical Report MSR-TR-2016-1001, October 2016.

[44] Felix Gessert, Florian Bücklers, and Norbert Ritter. Orestes: A scalable database-as-a-service architecture for low latency. In *2014 IEEE 30th international conference on data engineering workshops*, pages 215–222. IEEE, 2014.

[45] Mainak Ghosh, Ashwini Raina, Le Xu, Xiaoyao Qian, Indranil Gupta, and Himanshu Gupta. Popular is Cheaper: Curtailing Memory Costs in Interactive Analytics Engines. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–14, 2018.

[46] Kishore Gopalakrishna, Shi Lu, Zhen Zhang, Adam Silberstein, Kapil Surlaker, Ramesh Subramonian, and Bob Schulman. Untangling cluster management with helix. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, New York, NY, USA, 2012. Association for Computing Machinery.

[47] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. Amazon Redshift and the Case for Simpler Data Warehouses. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1917–1923, 2015.

[48] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, Boston, MA, March 2011. USENIX Association.

[49] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8, 2010.

[50] S Idreos, F Groffen, N Nes, S Manegold, S Mullender, and M Kersten. Monetdb: Two decades of research in column-oriented database. *IEEE Data Engineering Bulletin*, 2012.

[51] Jean-François Im, Kishore Gopalakrishna, Subbu Subramaniam, Mayank Shrivastava, Adwait Tumbde, Xiaotian Jiang, Jennifer Dai, Seunghyun Lee, Neha Pawar, Jialiang Li, et al. Pinot: Realtime OLAP for 530 Million Users. In *Proceedings of the 2018 International Conference on Management of Data*, pages 583–594, 2018.

[52] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billionscale similarity search with gpus. *IEEE Transactions on Big Data*, 7(3):535–547, 2021.

[53] Rajesh K Karmani, Amin Shali, and Gul Agha. Actor frameworks for the jvm platform: a comparative analysis. In *Proceedings of the 7th International Conference*

on Principles and Practice of Programming in Java, pages 11–20, 2009.

[54] Xavier Léauté. Benchmarking Druid. 2014.

[55] Sangmin Lee, Zhenhua Guo, Omer Sunercan, Jun Ying, Thawan Kooburat, Suryadeep Biswal, Jun Chen, Kun Huang, Yatpang Cheung, Yiding Zhou, Kaushik Veeraraghavan, Biren Damani, Pol Mauri Ruiz, Vikas Mehta, and Chunqiang Tang. Shard manager: A generic shard management framework for geo-distributed applications. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 553–569, New York, NY, USA, 2021. Association for Computing Machinery.

[56] Roman Leventov. The Challenges of Running Druid at Large Scale, Nov 2017.

[57] Yi Lin, Bettina Kemme, Marta Patiño-Martínez, and Ricardo Jiménez-Peris. Middleware Based Data Replication Providing Snapshot Isolation. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 419–430, 2005.

[58] Barbara Liskov, Atul Adya, Miguel Castro, Sanjay Ghemawat, R Gruber, U Maheshwari, Andrew C Myers, Mark Day, and Liuba Shrira. Safe and efficient sharing of persistent objects in thor. *ACM SIGMOD Record*, 25(2):318–329, 1996.

[59] Michael McCandless. Lucene nightly benchmarks. 2020.

[60] Gian Merlino. Druid Initial Join Support, Oct 2019.

[61] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577, Carlsbad, CA, October 2018. USENIX Association.

[62] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 439–455, New York, NY, USA, 2013. Association for Computing Machinery.

[63] Marta Patiño-Martinez, Ricardo Jiménez-Peris, Bettina Kemme, and Gustavo Alonso. MIDDLE-R: Consistent Database Replication at the Middleware Level. *ACM Transactions on Computer Systems (TOCS)*, 23(4):375–423, 2005.

[64] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[65] Mark Raasveldt. MonetDBLite: An Embedded Analytical Database. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1837–1838, 2018.

[66] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10. Ieee, 2010.

[67] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. E-Store: Fine-Grained Elastic Partitioning for Distributed Transaction Processing Systems. *Proceedings of the VLDB Endowment*, 8(3):245–256, 2014.

[68] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[69] Stephanie Wang, Eric Liang, Edward Oakes, Ben Hindman, Frank Sifei Luan, Audrey Cheng, and Ion Stoica. Ownership: A distributed futures system for fine-grained tasks. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 671–686. USENIX Association, April 2021.

[70] Fangjin Yang, Eric Tschetter, Xavier Léauté, Nelson Ray, Gian Merlino, and Deep Ganguli. Druid: A Real-Time Analytical Data Store. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 157–168, 2014.

[71] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Ulfar Erlingsson, Pradeep Gunda, and Jon Currey. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2009.

[72] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Presented as part of the*

9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12), pages 15–28, San Jose, CA, 2012. USENIX.

[73] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page

423–438, New York, NY, USA, 2013. Association for Computing Machinery.

[74] Siyuan Zhou and Shuai Mu. Fault-tolerant replication with pull-based consensus in mongodb. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 687–703. USENIX Association, April 2021.

Orca: Server-assisted Multicast for Datacenter Networks

Khaled Diab   Parham Yassini   Mohamed Hefeeda

School of Computing Science
Simon Fraser University
Burnaby, BC, Canada

Abstract

Group communications appear in various large-scale datacenter applications. These applications, however, do not currently benefit from multicast, despite its potential substantial savings in network and processing resources. This is because current multicast systems do not scale and they impose considerable state and communication overheads. We propose a new architecture, called Orca, that addresses the challenges of multicast in datacenter networks. Orca divides the state and tasks of the data plane among switches and servers, and it partially offloads the management of multicast sessions to servers. Orca significantly reduces the state at switches, minimizes the bandwidth overhead, incurs small and constant processing overhead, and does not limit the size of multicast sessions. We implemented Orca in a testbed to demonstrate its performance in terms of throughput, consumption of server resources, packet latency, and the impact of server failures. We also implemented a sample multicast application in our testbed, and showed that Orca can substantially reduce its communication time, through optimizing the data transfer between nodes using multicast instead of unicast. In addition, we simulated a datacenter consisting of 27,648 hosts and handling 1M multicast sessions, and we compared Orca versus the state-of-art system in the literature. Our results show that Orca reduces the switch state by up to two orders of magnitude, the communication overhead by up to 19X, and the control overhead by up to 14X, compared to the state-of-art.

1   Introduction

Many modern datacenter applications require group communications in the form of one-to-many or many-to-many patterns. Examples of these applications include distributed databases, telemetry systems, consensus protocols, and machine learning systems. Multicast can efficiently support these communication patterns. For example, in distributed databases, multicast can be used to distribute and replicate data among servers [69]. For telemetry systems, multicast is

suitable for sending updates and monitoring data to collector nodes [46,67]. In addition, multicast can be used for state machine replication tasks in the Paxos consensus protocol and its variations [21,39,45,53]. Furthermore, multicast can improve the performance of iterative algorithms that distribute data from a server to multiple working nodes. Examples of such algorithms appear in training machine learning models [28], text mining [48], and recommendation systems [36].

In addition to the above applications, an efficient multicast primitive would benefit various systems that naturally perform group communication. For example, publish-subscribe systems [37,58,68] typically send each message to a group of receivers. These systems are the substrate for many applications such as activity trackers, log aggregators, and stream processing frameworks. Moreover, in the emerging serverless platforms [2,60], a common pattern is that a worker communicates with multiple other workers to enroll them in a single burst computation [7, 8], which can efficiently be realized using multicast.

Despite its potential significant bandwidth savings, multicast faces multiple challenges that slow down its deployment by major cloud providers [62]. First, to forward packets on links belonging to the multicast tree, multicast forwarding requires *maintaining state at all switches* for each session, which imposes substantial memory overheads on switches. Second, *updating and refreshing this state* upon changes generates a storm of messages, which reduces the scalability of switches as they are required to process numerous control packets. Finally, since multicast trees in datacenters could potentially span many switches and servers, *encoding these trees into labels* could impose substantial processing, communication and/or bandwidth overheads.

As a result, there has been a lack of efficient and scalable multicast systems that support large numbers of sessions. For example, in practice, switch vendors are forced to limit the number of IP multicast sessions per switch [55], because of the inefficiencies introduced by the group management [50] and tree construction [16] protocols of IP multicast. This is not cost-effective for cloud providers. In addition, while

current datacenter multicast approaches, e.g., [5, 6, 32, 43], improve upon the basic IP multicast, they also do not scale well and impose substantial overheads on the network, as we show in this paper. To partially mitigate the lack of efficient multicast systems, many datacenter applications had to rely on (inefficient) application-layer protocols [51]. For example, Apache Spark [40] implements its own primitives [9] such as Cornet [36] and HTTP-based multicast.

This paper presents a new architecture, called Orca, to realize efficient multicast forwarding that can support millions of concurrent multicast sessions in datacenter networks. The idea of Orca is to *offload* some of the state maintained at network switches to end servers. To achieve this idea, Orca computes *fixed-size and compact* labels and attaches them to packets of multicast sessions. These labels effectively enable shifting some of the data plane tasks to servers. As a result, Orca significantly reduces the state at switches, minimizes the bandwidth overhead, incurs small and constant processing overhead, does not limit the size of multicast sessions, and eliminates redundant traffic. Realizing the proposed *server-assisted* multicast approach, however, faces multiple challenges at the control and data planes that Orca addresses. At the control plane, the proposed architecture needs to calculate optimized labels, manage state at servers, and handle failures. At the data plane, it requires packet processing algorithms at switches and servers that sustain the line-rate performance and minimize the latency and resource consumption.

This paper makes the following contributions.

- We introduce the idea of *server-assisted (or offloaded) multicast* for scalable multicast services in datacenters.

- We design a hierarchical control plane that efficiently manages multicast sessions and their dynamics, handles network failures, and does not impose high control overheads (§3.3 and §3.4).

- We present a scalable data plane algorithm to process multicast packets within high-speed datacenter networks, without introducing redundant traffic or requiring switches to maintain large states (§3.5).

- We design and implement APIs to transparently integrate multicast into datacenter applications (§4).

- We implement the proposed multicast approach and evaluate its performance in a testbed using programmable switches to demonstrate its practicality (§5). Our results show that the proposed approach can support high-speed traffic, uses small CPU resources at servers, and imposes small and predictable packet delays.

- We show the potential significant gains achieved by using multicast instead of unicast in datacenter applications. We implemented a sample application using Orca and the unicast approach used in current systems such as Apache Spark [40]. For this application that has only 12 receivers, our results show that Orca can reduce the communication time by almost an order of magnitude; larger savings are expected for applications with more receivers. In addition, since an Orca sender transmits only one copy per packet regardless of the number of receivers in the session, the required CPU resources are significantly reduced, compared to using unicast.

- We compare Orca against the closest system in the literature, Elmo [5], in large-scale simulations (§6). Our results show that Orca reduces the switch state by up to two orders of magnitude, the communication overhead by up to 19X, and the control overhead by up to 14X compared to Elmo in large-scale datacenter networks.

2 Related Work

Internet Multicast. IP multicast is not practical for datacenter networks because of its limited scalability for both the control and data planes [11, 52]. Specifically, its group management and tree construction protocols, e.g., IGMP [50] and PIM [16], need to maintain state at routers belonging to each multicast session. Moreover, to refresh this state, these protocols generate control messages that routers need to process. These overheads limit the number of multicast sessions, and they could delay a receiver joining a session for up to 23 seconds [66], which is not practical for datacenters. Furthermore, current multicast approaches designed for ISP networks, e.g., [1, 43], introduce significant communication overheads, and thus they are not suitable for datacenter networks.

Datacenter Multicast. Multiple approaches, e.g., [5, 32, 35], attempted to address the challenges of IP multicast. Li et al. [35] propose a multi-class Bloom filter (MBF) to support multicast in datacenter networks. For every interface, MBF uses a Bloom filter to store whether packets of a session should be duplicated on that interface. MBF may introduce redundant traffic due to the probabilistic nature of Bloom filters. Li and Freedman [32] partition the IP address space and aggregate addresses for different sessions. However, this approach consumes the limited flow table resources in switches and limits the number of supported multicast sessions. Elmo [5] encodes links of a multicast tree into rules to be attached to packets and maintained at switches. Elmo is the state-of-art multicast system for datacenters, and we compare against it.

Other works, e.g., [9, 10, 25], enabled multicast in circuit-switched datacenter networks. For example, Republic [9] and Blast [25] realize multicast by using additional optical circuit switches. Orca is designed to be deployed in the common packet-switched networks. Application-layer multicast approaches could also be used in datacenters, by concurrently sending unicast flows to multiple receivers. This, however, results in inefficient bandwidth utilization [47, 49, 51], and increases the CPU load on the sender.

Server-assisted Data Planes. While Orca is the first server-assisted multicast for datacenters, to the best of our knowledge,

it is not the first work to utilize server resources to implement parts of the data plane. For examples, Katta et al. [17] propose an OpenFlow [26] rule caching system using both switch TCAM and server memory, which is managed by a controller. In contrast, we design Orca to reduce the state maintained at switches instead of improving how the large number of rules are stored. A recent work [4] offloads the state of network functions to the server memory using RDMA. Unlike this system, Orca has small header sizes and its agents maintain small state instead of large network function state. Moreover, Orca simplifies how state is fetched, managed, and replicated.

3   Orca: Server-assisted Multicast

We start this section by specifying the design goals of Orca. Then, we present an overview of Orca describing its main components and how they work together. This is followed by the details of each component. In the **Appendix**, we describe various overheads, extensions, and limitations of Orca.

3.1   Design Goals

The objective of this paper is to design a multicast architecture for datacenter networks that achieves the following goals:

- *Reduce State at Switches*. Maintaining large state at network switches not only consumes their scarce memory resources, but it also increases the number and frequency of exchanged update messages to handle network failures and session dynamics. This forces switches to process many control messages while forwarding data packets, which may slow down the data plane [66].

- *Minimize Communication Overhead*. We aim at minimizing the header size per packet to reduce the communication (or bandwidth) overhead, which is critical to decreasing the total transmission time. We note that some of the existing multicast systems, e.g., [5], attach labels that can be as large as the packet payload.

- *Support Large-scale Multicast Sessions*. As datacenter applications become complex, the numbers of multicast sessions and receivers per session are expected to grow at high rates [70]. Existing systems, e.g., [32], do not efficiently scale to support the growing demands and high dynamics of recent datacenter applications.

- *Avoid Redundant Traffic*. Switches should forward packets *only* on links belonging to the multicast tree. This is because redundant traffic wastes network resources and overloads switches. Many of the existing multicast systems, e.g., [5, 35], cannot avoid sending redundant traffic without imposing a substantial amount of communication overheads by using large label sizes and/or increasing the state size maintained at switches.

Simultaneously realizing these goals is challenging as they are inter-dependent and pose conflicting trade-offs. For example, although attaching a large label to packets reduces switch state, it significantly increases the communication overhead and packet processing at switches. Our approach to concurrently achieve these design goals is to attach a *small* and *fixed-size* label to packets of every multicast session. This substantially minimizes the communication overhead and reduces packet processing on switches. In addition, we carefully calculate and process labels to eliminate redundant traffic. Furthermore, to reduce state at switches, we make servers assist in forwarding the packets. As a result, switches will be able to support large-scale multicast sessions.

3.2   Overview

Orca is designed for multi-rooted Clos topologies that are widely deployed in datacenter networks. We use the leaf-spine topology throughout the paper, but the same principles apply for other tree-based topologies. In the leaf-spine topology, the top layer consists of core switches that connect different leaf-spine planes. Spine switches connect leaf switches to other leaf switches and to core switches. Every leaf switch connects a rack of servers to the datacenter network. Each server runs a hypervisor switch and hosts multiple virtual machines (VMs).

In traditional IP multicast, network switches need to maintain state about each multicast session, which imposes significant overheads on the switches. In contrast, the proposed approach *carefully offloads* most of the work needed to manage multicast sessions to end hosts in the datacenter, which enables efficient and scalable multicast–a long standing problem. In addition, unlike IP multicast, Orca uses labels to direct the forwarding of multicast packets through the network. Each label consists of different components, each of which encodes a specific datacenter layer (i.e., leaf, spine, or core). However, as the size of a multicast tree grows, simple stacking of label components would lead to large, *variable-size*, labels and thus significant communication and processing overheads.

The proposed architecture is based on three key insights that enable us to design *small* and *fixed-size* labels and achieve scalability. First, a large portion of the label overhead comes from encoding the tree downstream links belonging to leaf switches, and that this overhead increases for multicast trees with large numbers of receivers. Second, labels belonging to leaf downstream links are not needed until the packet reaches a leaf switch. Third, servers in datacenters already host hypervisor switches to process various packet types. Based on these insights, we logically divide the data plane at the leaf layer: between each leaf switch and the servers connected to it. Then, we offload handling of the leaf downstream labels to some of the servers. To process the labels, these servers run an *Orca agent*, which can run on SmartNICs or CPU cores by integrating it with an existing hypervisor switch or running it as a standalone process.
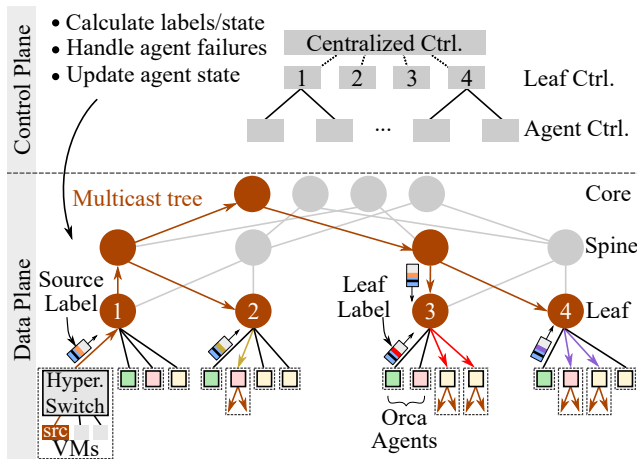
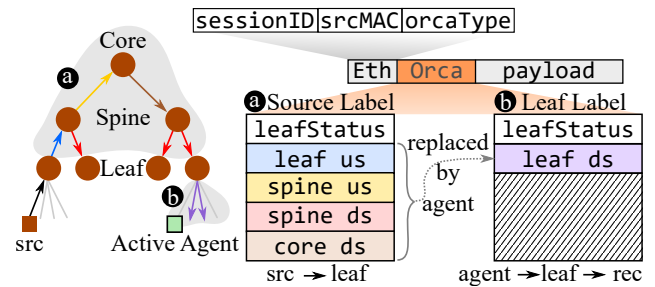Figure 1: The proposed multicast architecture.

Figure 2: Structure of the Orca header. The color of each label component matches the corresponding link in the network.

3.3 Calculating Labels

Labels play a critical role in the proposed multicast architecture, and they need to be carefully designed to ensure proper functioning of the multicast system as well as minimize the communication and processing overheads.

The centralized controller computes a fixed-size source label that consists of *four components* and a single `leafStatus` bit. Figure 2 illustrates the header format of an Orca data packet and the label structure. The four label components encode tree links belonging to leaf upstream (us), spine upstream, spine downstream (ds), and core downstream links. When a data packet reaches an active agent, the source label is replaced with the corresponding leaf label to forward packets to the multicast receivers.

The centralized controller calls the CALCULATELABELS algorithm (pseudo code is given in Algorithm 1) to calculate a source label to be attached to packets of the multicast session, a set of leaf labels to be maintained at the agents and state to be maintained at spine switches (if needed). No session state is needed at core or leaf switches. The algorithm takes as input the multicast tree $\mathbb{T}$. It groups tree links of each network layer and encodes their IDs independently in a fixed-size label component.

The algorithm first creates a bitmap of size (in bits):
$$1 + \max(L_d, \lceil \log(L_u) \rceil + P_d + F + \lceil \log(P_u) \rceil + C_d),$$

where L_d, L_u, P_d, P_u, and C_d are the maximum numbers of downstream and upstream ports per leaf switch, downstream and upstream ports per spine switch, and downstream ports per core switch. F is the size of a filter encoding the spine downstream links. This bitmap accommodates the leaf labels that will be inserted by agents. A typical datacenter switch has 48 ports [5]. Thus, the size of Orca label is 19 bytes in most practical cases.

The first bit in an Orca source label is the `leafStatus` bit, which indicates whether an agent has replaced a source label with a leaf label. The remaining bits are used to encode links of the multicast tree based on four cases as follows.

Case 1: Leaf and Spine Upstream. For the leaf and spine upstream links, the CALCULATELABELS algorithm maps the two link IDs to outgoing ports in the leaf and spine

As illustrated in Figure 1, Orca is composed of two components: (i) Hierarchical Control Plane and (ii) Server-assisted Data Plane. The Control Plane is composed of three controllers: Centralized, Leaf, and Agent. The Centralized controller creates labels to represent multicast trees and decides the state that needs to be maintained at network switches; details are presented in §3.3. A leaf controller is deployed on each leaf switch, while agent controllers are deployed on VMs within racks. All three components of the Hierarchical Control Plane collaborate to handle network and agent failures as well as to manage the dynamic nature of multicast sessions, as described in §3.4. The Data Plane, presented in §3.5, instructs switches and Orca agents how to process packets.

At a high-level, a multicast session is created and managed as follows, refer to Figure 1. The tree spanning the source and receivers has one path from the source VM to any core switch, then it reaches the receivers by branching to spine and leaf switches. The tree is then given to the centralized controller, which creates a fixed-size label (referred to as *source label*) to represent a part of the tree. It is important to notice that although the multicast tree can be large and spans many parts of the datacenter network, Orca optimizes the source label and keeps its size small and constant, as described in §3.3. The source label is sent to the source of the session, which attaches it to each packet. The packet is then sent upstream to spine and core switches, which forward it based on various components of the source label in that packet. Then, the packet is sent downstream from the spine and core switches, using other components of the source label, to the leaf switches that have receivers of the session in their racks. Each leaf switch sends the packet to an active Orca agent within its rack. The agent replaces the source label with another label (called *leaf label*) and sends it back to the leaf switch. The leaf label contains the information needed by the leaf switch to forward the packet to the end receivers within the rack.

Algorithm 1 The CALCULATELABELS algorithm.

Input: $\mathbb{T}$: multicast tree
Output: L: computed source label sent to the source VM
Output: $\mathbb{S}$: state sent to a subset of the spine switches
Output: $\mathbb{F}$: set of leaf labels, each is sent to an agent

```
 1: function CALCULATELABELS(T)
 2:     ⟨L, S⟩ = CALCULATESOURCELABEL(T)
 3:     F = CALCULATELEAFLABELS(T)
 4:     return ⟨L, S, F⟩
 5: function CALCULATESOURCELABEL(T)
 6:     size=1 + max(L_d, ⌈log(L_u)⌉ + P_d + F + ⌈log(P_u)⌉ +
        C_d)
 7:     L = BitString(size)
 8:     L.append(0) // Set leafStatus to 0 (source label)
 9:     Case 1: Leaf and Spine Upstream.
10:     L.append(T.leaf_us_link().port_num)
11:     L.append(T.spine_us_link().port_num)
12:     Case 2: Spine Downstream.
13:     // Common downstream ports across spine switches
14:     C = FINDCOMMONPORTS(T.spine_switches())
15:     L.append(MAPTOBITSTRING(C, P_d))
16:     // Call Algorithm 2
17:     ⟨D, S⟩ = ENCSPINEDSLINKS(T, C, F)
18:     L.append(D)
19:     Case 3: Core Downstream.
20:     core_links = T.core_ds_links()
21:     L.append(MAPTOBITSTRING(core_links, C_d))
22:     return ⟨L, S⟩
23: function CALCULATELEAFLABELS(T)
24:     F = {}
25:     Case 4: Leaf Downstream.
26:     for (leaf ∈ T.leaf_switches()) do
27:         // Each bit set to 1 represents a session receiver
28:         lbl = MAPTOBITSTRING(leaf.ds_links(), L_d)
29:         F = F ∪ lbl
30:     return F
```

switches and encodes these port numbers as two labels of sizes $\lceil \log(L_u) \rceil$ and $\lceil \log(P_u) \rceil$ bits, respectively.

Case 2: Spine Downstream. Since the multicast tree may include more than one spine switch, reserving a bit per spine downstream link significantly increases the label size. Instead, we trade off large label sizes, which impose overhead on every single multicast packet, with a small state maintained at a subset of the spine switches. Specifically, we encode the spine downstream links using two label components with a total size of $P_d + F$ bits. The first label component encodes the common downstream ports across all spine switches belonging to the multicast tree using P_d bits. For example, if a tree has three spine switches and the first two outgoing ports belong to the tree for each of the three spine switches, then the calculated label is $1100\ldots0$. We refer to this set of common ports as $\mathbb{C}$.

The second label component uses a probabilistic set membership data structure (a.k.a *filter*) to encode the remaining spine downstream links in a label D of size F bits. Since these filters trade off membership accuracy for space efficiency, they may result in false positives, which occur when some spine downstream links that do not belong to the multicast tree are incorrectly included in the computed filter. False positives result in redundant traffic. To address this issue, we calculate a state alongside the label. This state can have zero or more entries, and each entry takes the form $\langle sID, linkID \rangle$, where sID is the ID of the spine switch that should maintain this state and $linkID$ is the ID of the downstream link identified as a false positive during the encoding. The filter supports two functions: (i) $D = encode(l)$ to encode an input item l (link ID in our case) into a bit string D of size F bits using a hash function, and (ii) $check(l, D)$ to check whether a given item l belongs to D using the same hash function. Our link encoding algorithm can use any filter, e.g., Bloom [54] and Cuckoo [27] filters, that can support: (1) adding an item to an existing filter, (2) testing whether an item exists (potentially with false positives), and (3) avoiding false negatives. A false negative happens when a link in the multicast tree is not represented in the filter.

The CALCULATELABELS algorithm calls the ENCSPINED-SLINKS function, the pseudo code is shown in Algorithm 2 in the Appendix. This function encodes spine downstream links of the multicast tree into a label D and calculates the state $\mathbb{S}$ to be maintained by spine switches. To calculate $\mathbb{S}$, we need to identify false positive links belonging to spine switches. We refer to the subset of the spine downstream links that *may* be false positives as *candidates*. There are two conditions for a spine downstream link to be a false positive candidate. First, it has to be attached to a spine switch that belongs to the multicast tree, as packets of that session do not reach other spine switches. Second, it should not belong to the spine downstream links of the multicast tree. Otherwise, it is not a false positive.

The ENCSPINEDSLINKS function has three steps. First, it encodes every link l in the set of spine downstream links using the *encode* function. Then, it computes the false positive candidates based on the two conditions mentioned earlier. Finally, it calculates the state that needs to be maintained at spine switches by checking all false positive candidates stored in *cands* and adding only the links that collide with the spine downstream links encoded in D and not belonging to $\mathbb{C}$.

Case 3: Core Downstream. The CALCULATELABELS algorithm maps IDs of core downstream tree links to a bitmap of size C_d bits, where C_d is the maximum number of downstream ports in the core layer. The label bits identify the outgoing ports at the core switch belonging to the multicast tree. Thus, a bit at location i in the label is set to 1 if the core switch should duplicate packets on the ith port.

Case 4: Leaf Downstream. For every leaf switch belonging to the multicast tree, the CALCULATELABELS algorithm

calculates a leaf label that encodes all link IDs to reach the receivers of the session within the rack managed by that leaf switch. Each leaf label simply maps the link IDs into a bitmap of size L_d bits.

3.4 Handling Session Dynamics and Failures

Orca employs simple, but effective, mechanisms to manage the dynamic nature of multicast sessions, and to mitigate network and agent failures. We only assume the continuous availability of the top-level, centralized controller of Orca, which can be achieved through mechanisms usually used for such datacenter functions, e.g., [42].

Session Dynamics. Multicast receivers can join and leave any time during the sessions by calling corresponding APIs that communicate with the centralized controller (§4).

When a joining/leaving event is received, the centralized controller runs a simple method (Case 4 in §3.3) to update leaf labels at the agents. The controller then sends the updated leaf labels to the corresponding leaf controllers. The message also includes a unique sequence number. Each leaf controller relays the new leaf label to all active and standby agents within its rack. An agent updates its memory with the new label if the received sequence number is larger than the largest sequence number it has processed so far. Agents then send confirmation messages to upstream controllers indicating that the new changes were processed successfully.

Orca Agent Failures. In each rack, we maintain N Orca agents active and M as standby, where N, M are configurable parameters. All agents within a rack maintain the same leaf label per multicast session. The leaf switch in the same rack distributes the labeling workload among the N active agents, in a round robin manner. This adds more reliability and reduces the labeling load on individual agents.

All agents, active and standby, send heartbeat packets to the leaf controller at a fixed rate. If the leaf controller does not receive any heartbeats from an agent within a timeout period T, the agent is assumed failed. T is in the same order of the RTT within a single rack, which is often a few milliseconds [19]. If the failed agent was active, the leaf controller replaces it by one of the standby agents, otherwise the controller just removes the failed agent from the standby set.

We note that Orca agents deployed in a rack operate independently of agents deployed in other racks. Thus, our approach *localizes* failures within each rack, which reduces the control overhead and increases the control plane responsiveness. In other words, a leaf controller handles *only* the failures of its downstream agents. In addition, heartbeats provide responsiveness and simplicity, which is sufficient in our system as all agents maintain the same state. The state is updated across all agents when there is a change in the multicast tree, which is detected by the centralized controller.

Network Failures. The centralized controller detects network (link and switch) failures using existing systems such as [12].

Once a failure is detected, the controller re-calculates new source and leaf labels for the impacted sessions (§3.3). It also computes a new state at switches (if needed). To mitigate losses during network or agent failures, applications can use reliable transport protocols, e.g., [9].

3.5 Server-assisted Data Plane Forwarding

The data plane in Orca consists of leaf, spine and core switches, as well as agents deployed at servers. The data plane components process received packets as described below.

Leaf Switch. For a packet received on a downstream port, the leaf switch data plane processes that packet based on the `leafStatus` bit. If this bit is zero, i.e., a packet from the source, the data plane reads the first $\log(\lceil L_u \rceil)$ bits after the `leafStatus` bit as a leaf upstream label component, and forwards the packet based on the upstream port number encoded in that component. If the `leafStatus` is set to 1, this means the active agent has inserted a leaf label into the packet header. Thus, the data plane uses the leaf label component of size L_d bits to forward/duplicate the packet to corresponding servers. Specifically, a bit set at location i instructs the data plane to duplicate the packet on its ith port.

If a packet is received on an upstream port, the data plane forwards the packet on a port connected to one of the active agents, which is set and updated by the leaf controller.

Spine Switch. For a packet received on a downstream port, the data plane processes both the upstream and downstream label components. First, the packet is forwarded to a core switch by reading the spine upstream label, which encodes the outgoing port number. Second, since the packet may be forwarded/duplicated on the spine downstream links, the data plane runs the PROCSPINEDSLABEL algorithm to process the two spine downstream labels (pseudo code is shown in Algorithm 3 in the Appendix). This algorithm is executed for packets received on upstream ports as well. The algorithm first identifies the common links $\mathbb{C}$ by reading the first label. If a link is set to one in the label, the switch duplicates the packet on that link. Then, the algorithm uses the second label D and state *State* maintained by the spine switch to decide which of the other downstream links belong to the tree.

For each link $l \notin \mathbb{C}$, the algorithm decides to not forward the packet on l if it is not encoded in D. This is because filters in Orca do not produce false negatives. When $l.id$ exists in the label, the algorithm needs to check the maintained state *State* as $l.id$ may be a false positive. Recall that the state contains the false positive links computed by the control plane. The algorithm duplicates the packet only if $l.id$ does not exists in $State[sID]$.

Core Switch. The data plane reads the core downstream label component (of size C_d bits) to forward/duplicate the packet to downstream spine switches. Similar to the leaf downstream label, this label encodes which downstream ports the incoming packet should be forwarded on.

Orca Agent. For incoming packets from a leaf switch, the agent data plane checks the `leafStatus` bit. If it is set to 0 (i.e., it has no leaf label), the agent reads the corresponding leaf label from the leaf label map and inserts it into the packet header and sets the `leafStatus` bit to 1. If the `leafStatus` bit is 1, this packet is destined to receiver VMs.

Overheads and Limitations of Orca. We describe Orca overheads and limitations in the Appendix. In summary, Orca agents require small processing resources at servers as the computation performed on packets is simple. In addition, Orca adds a small latency to packets at the leaf layer only. Furthermore, deploying Orca in graph-based datacenter networks requires changes in some of its components.

4   Implementation and Orca APIs

We briefly describe the implementation of Orca components which are illustrated in Figure 3.

Orca APIs for Multicast Applications. We implemented two sets of interfaces for multicast applications. The first one is between the agent and applications to provide `send` and `recv` functionalities seamlessly to the application. These APIs use Unix domain sockets to communicate with the agent. When using `send` at the source, the agent gets data from the sockets, attaches Orca label and transmits the packets to the leaf switch. Receivers use `recv` to instruct the agent to relay available data to the application. The second set of APIs is between applications and the centralized controller to `create`, `join` and `leave` multicast sessions. We implemented the communication and data encoding/decoding using gRPC [64] and Protocol Buffers.

Orca Agents. We implemented the agent using BESS [23,61] in about 640 lines of C++, where packet processing is done completely in the user space using DPDK [63]. The agent leverages Receive Side Scaling (RSS) to receive packets on different RX queues, each is assigned to a single core.

Orca Hierarchical Controller. The centralized and leaf controllers are implemented in about 3K lines of Golang. The current implementation of the leaf controller communicates with the data plane through raw or Unix domain sockets, but it can easily support other interfaces. For instance, in our testbed, the leaf controller process is deployed to the workstation hosting a NetFPGA, and it uses PCIe to exchange control packets with the NetFPGA, which is done through the RIFFA framework [18]. Communications between the centralized and leaf controllers are done using gRPC [64].

Switch Data Plane. Since Orca's data plane processing is simple, it can easily be implemented in different programmable switches. We implemented the data plane of Orca in NetFPGA SUME [29] and tested it on multiple of them. We used the open source project in [56], and implemented a Verilog module to decide the outgoing ports. We measured the number of clock cycles and resource usage of our implementation using Xilinx tools. Our implementation of core and leaf
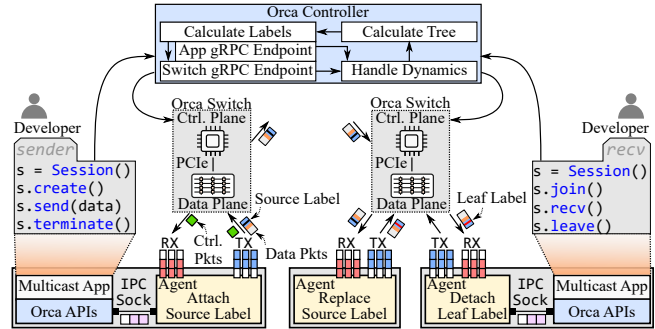

Figure 3: Implementation of Orca components.

switches maps the corresponding bitmaps to outgoing ports, which is done in one clock cycle. We implemented the spine switch algorithm in three clock cycles to identify common ports and check the Bloom filter using a bitwise-AND between the label and hashed link IDs stored at the switch, read state from memory, and decide the outgoing ports. In terms of resource usage, our algorithm utilizes a tiny percentage of the available hardware resources. It uses 0.12% and 0.16% of the available lookup tables (LUTs) and registers, respectively.

5   Evaluation of Orca in Testbed

We evaluate Orca in a testbed to demonstrate its potential benefits to applications and asses the performance of its data plane and control plane components. The testbed has three NetF-PGA SUME switches [29] representing a spine and two leaf switches, each of which has four 10GbE ports. The testbed also has five workstations to act as Orca agents and multicast senders and receivers. We configure our testbed to only have one active agent per rack to stress our labeling algorithm at servers. Each workstation is equipped with a dual-port Intel 82599ES 10GbE NIC. Each leaf switch is connected to two workstations, and the spine switch is connected to one workstation. We generate traffic at line rate from a multicast source and transmit it to leaf switches through the spine switch.

5.1   Benefits of Orca

We implemented a sample multicast application that has the same behavior of the iterative machine learning algorithms implemented in Spark [40]. In these algorithms, the data to be processed is often written to files, and a server iteratively sends them to all receivers for processing. In our application, a server reads a file and transmits it in chunks. In every iteration, after a file is sent, the server awaits acknowledgment of file reception from the receivers. The next round starts only after receiving all acknowledgments. This emulates the aggregation phase in distributed data processing frameworks [44], which indicates all workers have updated their model parameters. In our implementation, we set the payload size to the maximum

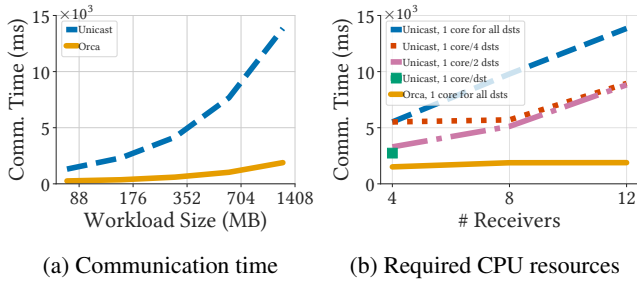(a) Communication time   (b) Required CPU resources

Figure 4: Benefits of Orca.

UDP message length. We run each instance of the client application inside a separate Docker container on the receiver machines.

Performance Metrics. We compare Orca versus the current unicast approach used in systems such as Apache Spark [40]. In particular, we demonstrate the potential benefits of Orca in terms of the communication time, impact of available processing capacity at the sender on the communication time, and total transmitted traffic.

The running time of datacenter applications consists of communication and computation times. The communication time of an application is the total time spent on sending data and receiving corresponding acknowledgments without including the computation times. We measure the communication time of Orca versus unicast, as this is the main aspect being optimized by Orca and it does not control or modify the computations. In addition, we aim at showing that Orca can present the same packet to the application layer much faster compared to the current unicast approaches.

Depending on the application and its total computation time, Orca can reduce the running times of a variety of applications. For example, the authors in [36] reported that the communication time of data-intensive tasks using unicast can be larger than the computation time, especially as the number of workers increases. Thus, optimizing data transfer is critical for these applications.

Workloads. The number and size of the transmitted files are similar to the ones used in the distributed latent Dirichlet allocation (LDA) algorithm [9, 30]. LDA identifies topics in the input documents and maps each document to a set of topics. The vocabulary training set is the data transmitted to the worker nodes. To calculate the workload size, we run the algorithm on a synthetic dataset containing 16,923 documents and 100 topics using the tool in [71]. To evaluate Orca using realistic workloads, we create five different workloads with sizes of 88MB, 176MB, 352MB, 704MB, and 1.4GB.

Results. We conduct experiments using concurrent 4, 8 and 12 receivers and the five different workloads mentioned above.

Figure 4a shows the communication time for Orca and unicast for different workloads when the number of receivers is 12. The sender in the multicast session uses one CPU core to transmit the traffic. These results show that Orca can

significantly reduce the communication time for all considered workloads. In addition, the figure shows that, unlike the case for Orca, the communication time for unicast grows in a super-linear manner with the workload size. This is because the unicast sender needs more time to transmit packets to each of the concurrent 12 receivers, whereas Orca transmits only a single packet for all receivers. Packet transmission at high rates also requires processing cycles.

To analyze the impact of the available processing capacity at the sender on the communication time, we allocate a varying number of CPU cores to transmit the traffic of the multicast session in the case of unicast. For Orca, only one CPU core is used. In Figure 4b, we plot the communication time for the largest workload (1.4GB) for Orca and unicast, as we vary the number of available CPU cores. The figure shows that Orca has a fairly stable communication time as the number of receivers increases, despite using only one CPU core to transmit all packets of the session. In contrast, unicast needs more CPU cores to send the traffic to different receivers to reduce the communication time. In our testbed, allocating a single core per receiver for unicast could not sustain the high packet rate at the sender for 8 and 12 receivers.

Next, we measure the total transmitted traffic from the sender for the largest workload as well as the label overhead of Orca. When using Orca, the total outgoing traffic is only 1.51 GB, compared to 18.01 GB when using unicast. This means the sender in the unicast model would need to transmit 12X more traffic, which not only consumes more bandwidth, but also requires more processing and memory resources to transmit much more packets. The total label overhead of Orca is 7.69 MB which represents only 0.51% of the transmitted multicast traffic.

Although current multicast approaches may yield similar benefits to applications, they cannot scale well to support a large number of multicast sessions. Therefore, we compare the scalability of Orca versus the state-of-art multicast system using large-scale simulations in §6.

Summary: For a sample application with 4–12 receivers, Orca achieves substantial savings in communication time, required processing resources at the sender, and bandwidth, compared to the current unicast approach.

5.2 Data Plane Performance

Throughput of Spine Switches. We report the throughput of the spine switch; we omit the results of leaf and core switches as they run simple forwarding algorithms.

We transmit labelled packets of many concurrent multicast sessions at the maximum link speed (i.e., 10 Gbps) from the source to the spine switch. The labels instruct the spine switch to duplicate packets to two leaf switches. We run this experiment five times for every packet size and compute the average across them. We compare the incoming packet rate against the outgoing packet rates observed at the two leaf
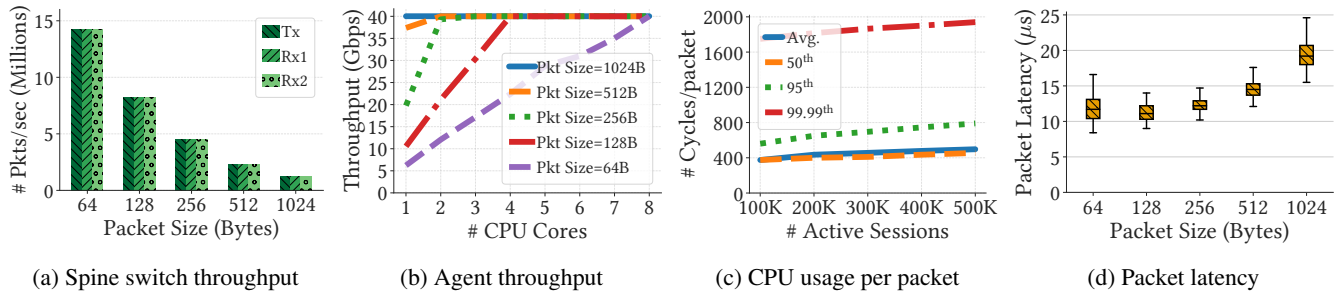
Figure 5: Data plane performance of Orca.

switches in Figure 5a. Our results show that the packet rates are the same (i.e., no packet losses). We also measure the achieved throughput at the two interfaces, and confirm that the spine switch can sustain the 10 Gbps for all packet sizes.

Agent Scalability. We stress and evaluate the scalability of the agent data plane. In this setup, we deploy two NICs (i.e., 4x10GbE ports) at the agent workstation and direct labeled traffic at rate of 40Gbps from the other two workstations. The labels have the `leafStatus` set to zero, which indicates that the agent needs to label them using a corresponding leaf label. We measure the throughput of the packets after being processed by the agent.

Figure 5b shows the throughput versus the number of allocated cores for the data plane, which shows that the agent scales well to support high rates. We measure the smallest packet size at which the agent can sustain the 40Gbps traffic using a single core. Our results show that the agent can sustain this rate using 1 core for packets of size 560 bytes or larger. For enterprise datacenters, the average packet size is reported to be 850 bytes [41]. Furthermore, data-intensive jobs like Hadoop workloads often use 1500-byte packets [20]. That is, Orca agents require only a few cores per rack to support many multicast sessions at high rates. Major datacenters deploy 24–48 servers per rack [14, 65], and each typically has more than 16 cores. That is, even for applications that require small 64-byte packets and send at an aggregate rate of 40Gbps, an Orca agent would need up to 1–2% of the available CPU resources in a rack when SmartNICs are unavailable.

In addition, recall from §5.1 that Orca requires only one CPU core at the sender side regardless of the number of receivers, whereas the current unicast approach needs a proportional number of CPU cores to sustain the transmission rate especially as the number of receivers increases. Thus, the processing capacity needed to run Orca agents will likely be offset by the savings in the processing capacity needed to transmit the traffic in the unicast approach.

Agent CPU Usage. Recall that an active agent needs to look up a leaf label from its memory using the session ID. We measure the total number of CPU cycles needed by the agent to process packets (including labeling and memory lookup). We stress the agent by allocating leaf labels for 1M sessions at

the agent. In this experiment, the sender randomly transmits traffic belonging to a subset of the total sessions, which we refer to as active sessions. We use large numbers of active sessions to stress the agent.

Figure 5c shows different statistics of the used number of CPU cycles per packet (measured by `rdtsc`) when the packet size is 1024 bytes. The results show the efficiency of the agent even without any code optimizations. For example, the agent running on a 3.8GHz CPU needs an average of 99 ns per packet when the number of active sessions is 100K per rack. We note that the number of CPU cycles is constant for different packet sizes, since the agent processes fixed-size labels. To put these numbers in context, existing, optimized, software switches such as OVS [24] and PISCES [15] use 409 and 426 cycles/packet, on average, to handle IP packets, respectively. The average for Orca is 375 cycles/packet when handling 100K active sessions.

Packet Latency and Jitter. We measure the packet latency and jitter of Orca at the leaf layer, which is defined as the total duration from when a packet is sent to the leaf switch to the time it is received by a multicast receiver connected to that switch. We emulate a dynamic traffic scenario, where the sender starts transmitting traffic at 1Gbps and increases the sending rate with 1Gbps steps every 20 seconds until it saturates the link, and holds this transmission rate for another 20 seconds.

Figure 5d shows the packet latency for different packet sizes. For 64-byte packets, the median latency is 11.3 μs. The packet latency slightly increases for large packet sizes because of the increased transmission time. Notice that in latency-sensitive applications, where latency for individual packets are important, smaller packets are more prevalent [57] where Orca has short packet latency.

We next measure the packet inter-arrival jitter, which is calculated as the difference between the current packet delay and previous packet delay. Our results show that Orca imposes negligible variance in packet latency. The average and maximum inter-arrival jitter values for 1024-byte packets are 1.135 μs and 3.3 μs, respectively.

Summary: *The Orca data plane is scalable and can sustain high throughputs even with small packet sizes. Orca agents*

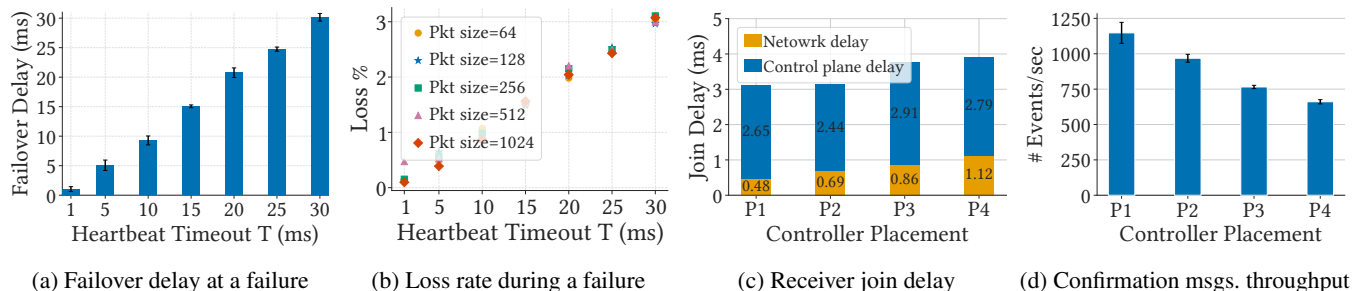| (a) Failover delay at a failure | (b) Loss rate during a failure | (c) Receiver join delay | (d) Confirmation msgs. throughput |

Figure 6: Control plane performance of Orca.

can process large numbers of concurrent sessions, use low CPU resources, and only add a small latency to packets.

5.3 Control Plane Performance

Responsiveness to Agent Failures. Orca localizes agent failures within the rack, thus we analyze failures for a single rack. We measure the performance while an active agent is handling traffic at 10Gbps. We manually crash the active agent and measure the following metrics at a receiver: failover delay, throughput, and data loss rate. We report the results for the worst-case scenario, where the rack has only one active agent. Failover delay is the time when the receiver does not receive traffic due to active agent failure and when it receives traffic again. Figure 6a shows the average failover delay for all packet sizes when we control the heartbeat timeout. The results confirm the fast response of the control plane in choosing a new active agent when the original agent fails. For instance, receivers can resume receiving traffic within 1.04 ms after an active agent fails when T is 1 ms.

We next measure the observed throughput at the receiver during a failure, where we crash the agent after two seconds. For 1024-byte packets and heartbeat timeout of 1ms, we observe a throughput drop by up to 0.012% only. In addition, for all packet sizes, the total throughput drop is less than 0.013% during a failure. Our results confirm that Orca quickly restores the transmission to full capacity after a failure. Finally, we plot the loss rate caused by an agent failure in Figure 6b. When the heartbeat timeout is 1 ms, Orca incurs a loss rate of 0.18%, on average, across all packet sizes. We note that such losses can be easily mitigated by using reliable multicast [9].

Receiver Joining Delay. We assess the performance of the proposed method for updating leaf labels when a new receiver joins. Recall that when a session changes, Orca sends new leaf labels to the corresponding agents. This impacts how quickly a joining receiver would receive traffic. In addition, network delays between the control plane components in datacenters might vary depending on the placement of the controllers and receivers. We emulate different controller placement setups in our testbed by adding synthetic delays at the network interface queues of the workstations (using `tc`) to stress our system.

We consider four different placement setups (P1–P4) starting with no synthetic delay in P1 with mean RTT of 479 μs and adding 200 μs of delay every step till we reach a mean RTT of 1,120 μs (maximum 1,326 μs) in P4 setup. These RTT values follow what is reported in [19] where the 99th percentile RTT between two hosts is 1.34 ms. We note that even the lowest RTT in our setup (i.e., P1) is larger than the median RTT inside a rack in datacenter networks which is 268 μs [19].

We measure the receiver join delay, which is the time duration from when a receiver sends a join request for a session and the time the first packet of that session is received by the receiver. For each placement setup, the experiment is repeated for 30 join events. Figure 6c shows the average join delay as well as the contribution of network delays. We report that even in the worst-case scenario (P4), the average join delay is less than 4 ms. In total, the median and 99th-percentile delays are 3.12 ms and 6.53 ms, respectively. To put these numbers into perspective, note that inserting a new rule into an OpenFlow switch takes 1–3 ms, and rule modification delays vary from 2–18 ms [22].

We next measure the throughput of processed confirmation messages at the centralized controller in Figure 6d, which represents the end-to-end performance. In the P1 setup, Orca handles an average of 1,147 msgs/sec (SD is 74). As increasing latency affects gRPC, the average throughput of the largest delay scenario is 662 msgs/sec (SD is 14).

***Summary**: Orca recovers quickly from failures and it supports dynamic multicast sessions. Furthermore, the failure detection mechanism in Orca is localized to individual racks.*

6 Orca versus State-of-Art

We analyze the performance and scalablity of Orca and compare it against the state-of-art system, Elmo [5], using large-scale simulations. We use the open-source code of Elmo.

Elmo employs three stages to encode a multicast tree. Elmo encodes switches of a tree as a union of multiple bitmaps representing outgoing ports. When the label size reaches a pre-configured value, Elmo installs forwarding state entries at switches without exceeding their capacities. Otherwise, Elmo calculates a default entry that may result in redundant traffic.
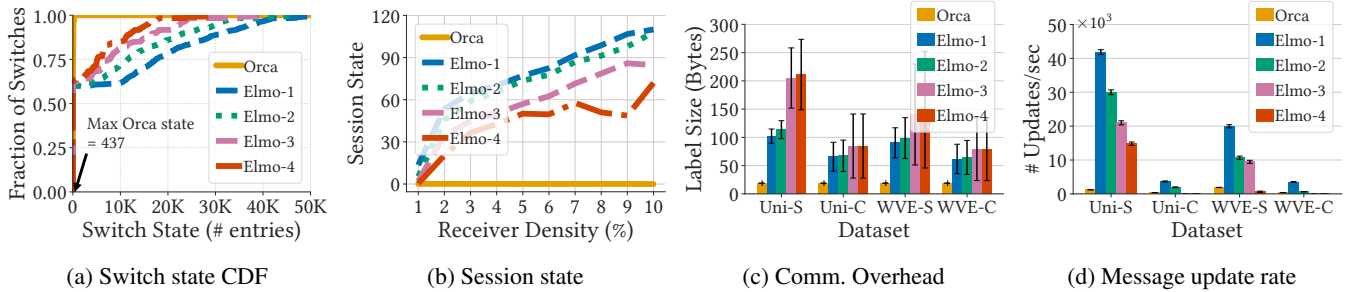
| (a) Switch state CDF | (b) Session state | (c) Comm. Overhead | (d) Message update rate |

Figure 7: Performance of Orca versus Elmo.

6.1 Simulation Setup

Topology and VM Placement. We simulate a multi-rooted Clos topology consisting of 48 pods, each has 576 48-port leaf and spine switches. This results in a large datacenter network of 27,648 hosts. We use a setup similar to [5, 32]: There are 3,000 tenants, each has a number of VMs ranging from 10 to 5,000. The maximum number of VMs per server is 20. We use two VM placement strategies. The first one is a Clustered placement (denoted by C), which places at most 12 tenant VMs per rack. The second is a Scattered strategy (denoted by S), and it places at most one tenant VM per rack.

Multicast Sessions and Datasets. Multicast receivers per session are randomly chosen from all tenant VMs. The size of these sessions follows two different distributions similar to [5, 32]. The first distribution follows a workload from the IBM WebSphere Virtual Enterprise (WVE) [32], and the second one is a uniform distribution (Uni). The minimum and maximum session sizes for both distributions are 5 and 5,000 receivers, respectively.

We generate four datasets representing various workload characteristics and VM placement strategies. We denote a dataset using its session size distribution and VM placement strategy, e.g., a dataset with uniform session sizes and scattered strategy is referred to as Uni-S. We simulate 1M multicast sessions per dataset, where each tenant has sessions proportional to the number of its VMs.

Orca and Elmo Parameters. We set the filter size in Orca to 69 bits to compute byte-aligned label. For Elmo, we control two parameters to analyze different aspects of it, and set them according to [5]. The first one is the number of rules encoded in Elmo label, which is set to be either 10 or 30. The second parameter is the redundancy limit that controls the amount of redundant traffic caused by sharing a single rule in Elmo label. We set this parameter to be 0 (no redundant traffic) or 12. We refer to the Elmo four configurations as Elmo-1 (10, 0), Elmo-2 (10, 12), Elmo-3 (30, 0) and Elmo-4 (30, 12).

6.2 Data Plane Performance

Switch State. Figure 7a shows the CDF of the switch state for the Uni-S dataset. The results for other datasets are similar.

The figure shows that Orca significantly reduces the state size compared to all considered configurations of Elmo. For example, in Orca, 99% of switches need to only maintain up to 253 entries in their memory, and no switch maintains more than 437 entries. In contrast, for Elmo-1, which calculates the smallest label sizes (i.e., 100 bytes on average), 99% of switches need to maintain up to 47.7K entries in their memory, with some switches need to maintain as many as 53.5K entries. Elmo could not reduce the state even when it doubles the label size. For example, in Elmo-4, 99% of switches need to maintain up to 24K entries in their memory (maximum is 30K entries). This is a significant improvement because it indicates that Orca requires much lower switch memory to support the same number of multicast sessions and much fewer control messages to update the switch state.

We next study the impact of session size on the required state to be maintained for that session in Figure 7b. The figure shows that Orca scales well, and it can reduce the session state by up to 55X compared to Elmo. For example, when a session has 2.5K receivers, Orca needs to maintain state at up to two switches only. Elmo-1, however, needs to maintain state at up to 110 switches.

These significant gains are achieved because, unlike Elmo, Orca does not require maintaining state at *any* leaf or core switch. In addition, the proposed spine labels can encode most of the spine downstream links while requiring small state at few spine switches.

Summary: *Orca reduces state size by up to two orders of magnitude compared to Elmo, and can support a large number of concurrent multicast sessions.*

Communication Overhead. Figure 7c shows that Orca reduces the communication overhead by using a small and fixed-size label of size 19 bytes to forward traffic of 1M sessions. On the other hand, Elmo uses much larger labels. For example, in the Uni-S dataset, the average and maximum label sizes of Elmo-4 are 211 bytes and 368 bytes, respectively; SD is 62 bytes.

Elmo introduces variations in the label size for the *same* configuration across different datasets. This means that changes in VM placement strategy or shifts in traffic patterns introduce unpredictable forwarding performance in Elmo, as

its switches need to process labels with *varying* sizes. For example, changing the VM placement strategy from clustered to scattered in Elmo-4 increases the average label size by 148% because receivers in the scattered strategy span more racks compared to the clustered one. For the same configuration, a shift in traffic pattern from WVE to Uni increases the average label size by 42% as Elmo needs to encode more receivers using the same label size. This is because the WVE distribution is skewed, and thus, fewer sessions have large group sizes compared to the Uni distribution. In contrast, Orca has a *fixed-size* label of 19 bytes because it utilizes the key insights described in §3.2.

Summary: *Orca reduces the communication overhead by up to 19X compared to Elmo while being robust against VM placement strategies and session sizes.*

Redundant Traffic. We define the redundant traffic per session as the ratio between the number of receivers that receive unwanted traffic to the total number of receivers in the session. *By design, Orca does not introduce any redundant traffic.* Elmo may introduce redundant traffic to reduce state size by controlling the redundancy limit parameter as it shares the same rule among multiple switches. We analyze the traffic redundancy of Elmo-2 and Elmo-4 for the Uni-S dataset. Other Elmo configurations have redundancy limit of 0 similar to Orca but they require maintaining much larger state at switches. Our results show that, for Elmo-2, 25% of the sessions have more than 67% redundant traffic, with a maximum value of 172%. For Elmo-4, with much larger labels, the maximum redundant traffic is 113%. That is, the traffic could erroneously be sent to more destinations not participating in the multicast session than the actual receivers.

Summary: *Orca does not introduce any redundant traffic, whereas Elmo may impose up to 172% redundant traffic.*

6.3 Control Plane Performance

Session Dynamics. We randomly generate 1,000 receiver joining/leaving events per second with joining probability of 0.5. Every event changes a multicast tree, and thus, the state maintained at switches may need to be refreshed. Refreshing the state requires the control plane to send update messages to the switches. We measure the total number of update messages per second sent by the controller for both Orca and Elmo. We report the results for all datasets in Figure 7d. For example, the Orca controller needs to send an average of 1,889 messages per second (SD is 45) for the WVE-S dataset. On the other hand, the Elmo controller needs to send 19.9K, 10K, 9.5K and 615 messages per second on average for Elmo-1, Elmo-2, Elmo-3 and Elmo-4, respectively. This is because Orca maintains state only at a small number of switches. Elmo-4 does not need to update many switches. It, however, imposes the largest label size among all Elmo configurations with high amount of traffic redundancy.

Summary: *Orca reduces the rate of update messages by*

up to 10X compared to Elmo.

Network Failures. Similar to session changes, a core or spine switch failure triggers Orca and Elmo to update state at switches if needed. For the WVE-S dataset, Orca needs to send 3,900 messages per core switch failure on average, while Elmo-1, Elmo-2, Elmo-3, and Elmo-4 needs to send an average of 56.2K, 31.2K, 27.6K, and 1.7K messages per core switch failure, respectively. For a spine switch failure, Elmo-1, Elmo-2, Elmo-3, and Elmo-4 send 34.7K, 20.6K, 18K, and 1.2K messages per failure, respectively, whereas Orca sends 4,890 messages per failure. Although Elmo-4 requires sending fewer messages per failure, it imposes significant overheads in terms of the label size and traffic redundancy.

Summary: *Compared to Elmo, Orca reduces the control overhead for handling failures by up to 14X.*

Running Time. Orca calculates labels faster than Elmo. We report the running time of Orca and Elmo in the Appendix.

7   Conclusions and Future Work

We presented Orca, an efficient multicast architecture for datacenter networks. Orca splits the data plane operations between leaf switches and servers. That is, Orca offloads managing multicast sessions from leaf switches to servers. Orca has a scalable control plane that handles session dynamics and network failures. It also has a simple data plane that can sustain high rates and can easily be implemented in programmable switches. The server component in Orca can be implemented on SmartNICs, or on regular CPU cores if SmartNICs are not available. We implemented lightweight APIs to seamlessly integrate multicast into datacenter applications. We also implemented Orca in a testbed that contains programmable switches. We evaluated a sample multicast application in our testbed. Our results show that Orca can substantially reduce the communication time compared to unicast. In addition, we assessed the performance of Orca in terms of its throughput, resource usage, packet latency and the impact of failures. Moreover, we compared Orca versus the state-of-art multicast system, Elmo, using large-scale simulations. Compared to Elmo, Orca reduces the switch state by up to two orders of magnitude and the label size by up to 19X.

This work can be extended in multiple directions. For example, we plan to extend Orca to support various group communication primitives needed by modern datacenter applications.

Acknowledgments

References

[1] Khaled Diab and Mohamed Hefeeda. Yeti: Stateless and generalized multicast forwarding. In *Proc. of USENIX NSDI'22*, Renton, WA, April 2022.

[2] Arjun Singhvi, Arjun Balasubramanian, Kevin Houck, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. Atoll: A scalable low-latency serverless platform. In *Proc. of ACM SoCC'21*, pages 138–152, Seattle, WA, November 2021.

[3] Stewart Grant, Anil Yelam, Maxwell Bland, and Alex C. Snoeren. Smartnic performance isolation with fairnic: Programmable networking for the cloud. In *Proc. of ACM SIGCOMM'20*, pages 681–693, Virtual Event, August 2020.

[4] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. Tea: Enabling state-intensive network functions on programmable switches. In *Proc. of ACM SIGCOMM'20*, pages 90–106, Virtual Event, August 2020.

[5] Muhammad Shahbaz, Lalith Suresh, Jen Rexford, Nick Feamster, Ori Rottenstreich, and Mukesh Hira. Elmo: Source-routed multicast for public clouds. In *Proc. of ACM SIGCOMM'19*, pages 458–471, Beijing, China, August 2019.

[6] Toerless Eckert, Gregory Cauchie, and Michael Menth. Traffic Engineering for Bit Index Explicit Replication (BIER-TE). Internet-draft, Internet Engineering Task Force, July 2019. Work in Progress.

[7] Collin Lee and John Ousterhout. Granular Computing. In *Proc. of ACM HotOS'19*, pages 149–154, Bertinoro, Italy, May 2019.

[8] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, João Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. February 2019. arXiv: 1902.03383.

[9] Xiaoye Steven Sun, Yiting Xia, Simbarashe Dzinamarira, Xin Sunny Huang, Dingming Wu, and TS Eugene Ng. Republic: Data multicast meets hybrid rack-level interconnections in data center. In *Proc. of IEEE ICNP'18*, pages 77–87, Cambridge, United Kingdom, September 2018.

[10] Xiaoye Steven Sun and TS Eugene Ng. When creek meets river: Exploiting high-bandwidth circuit switch in scheduling multicast data. In *Proc. of IEEE ICNP'17*, pages 1–6, Toronto, Canada, October 2017.

[11] O. Komolafe. Ip multicast in virtualized data centers: Challenges and opportunities. In *Proc. of IFIP/IEEE IM'17*, pages 407–413, Lisbon, Portugal, May 2017.

[12] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, and Alex C. Snoeren. Passive realtime datacenter fault detection and localization. In *Proc. of USENIX NSDI'17*, pages 595–612, Boston, MA, March 2017.

[13] Asaf Valadarsky, Gal Shahaf, Michael Dinitz, and Michael Schapira. Xpander: Towards optimal-performance datacenters. In *Proc. of ACM CoNEXT'16*, pages 205–219, Irvine, CA, December 2016.

[14] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. A cloud-scale acceleration architecture. In *Proc. of IEEE MICRO'16*, pages 1–13, Taipei, Taiwan, October 2016.

[15] Muhammad Shahbaz, Sean Choi, Ben Pfaff, Changhoon Kim, Nick Feamster, Nick McKeown, and Jennifer Rexford. Pisces: A programmable, protocol-independent software switch. In *Proc. of ACM SIGCOMM'16*, pages 525–538, Florianopolis, Brazil, August 2016.

[16] Bill Fenner, Mark J. Handley, Hugh Holbrook, Isidor Kouvelas, Rishabh Parekh, Zhaohui (Jeffrey) Zhang, and Lianshu Zheng. Protocol Independent Multicast - Sparse Mode (PIM-SM): Protocol Specification (Revised). RFC 7761.

[17] Naga Katta, Omid Alipourfard, Jennifer Rexford, and David Walker. Cacheflow: Dependency-aware rule-caching for software-defined networks. In *Proc. of ACM SOSR'16*, pages 1–12, Santa Clara, CA, March 2016.

[18] Matthew Jacobsen, Dustin Richmond, Matthew Hogains, and Ryan Kastner. Riffa 2.1: A reusable integration framework for fpga accelerators. *ACM Trans. Reconfigurable Technol. Syst.*, 8(4), September 2015.

[19] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proc. of ACM SIGCOMM'15*, pages 139–152, London, United Kingdom, August 2015.

[20] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. Inside the social network's (datacenter) network. In *Proc. of ACM SIGCOMM'15*, pages 123–137, London, United Kingdom, August 2015.

[21] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. Netpaxos: Consensus at network speed. In *Proc. of ACM SOSR'15*, pages 1–7, Santa Clara, CA, June 2015.

[22] Keqiang He, Junaid Khalid, Aaron Gember-Jacobson, Sourav Das, Chaithan Prakash, Aditya Akella, Li Erran Li, and Marina Thottan. Measuring control plane latency in sdn-enabled switches. In *Proc. of ACM SOSR'15*, pages 1–6, Santa Clara, CA, June 2015.

[23] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. Softnic: A software nic to augment hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015.

[24] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The design and implementation of open vswitch. In *Proc. of USENIX NSDI'15*, pages 117–130, Oakland, CA, May 2015.

[25] Yiting Xia, TS Eugene Ng, and Xiaoye Steven Sun. Blast: Accelerating high-performance data analytics applications by optical multicast. In *Proc. of IEEE INFO-COM'15*, pages 1930–1938, Hong Kong, China, April 2015.

[26] Diego Kreutz, Fernando M. V. Ramos, Paulo Esteves Veríssimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proc. of the IEEE*, 103(1):14–76, January 2015.

[27] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proc. of ACM CoNEXT'14*, pages 75–88, Sydney, Australia, December 2014.

[28] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *Proc. of USENIX OSDI'14*, page 583–598, Broomfield, CO, October 2014.

[29] Noa Zilberman, Yury Audzevich, G. Adam Covington, and Andrew W. Moore. NetFPGA SUME: Toward 100 Gbps as research commodity. *IEEE Micro*, 34(5):32–41, September 2014.

[30] Zhuhua Cai, Zekai J Gao, Shangyu Luo, Luis L Perez, Zografoula Vagena, and Christopher Jermaine. A comparison of platforms for implementing and running very large scale machine learning algorithms. In *Proc. of ACM SIGMOD'14*, pages 1371–1382, Snowbird, UT, June 2014.

[31] Dan Li, Mingwei Xu, Ying Liu, Xia Xie, Yong Cui, Jingyi Wang, and Guihai Chen. Reliable multicast in data center networks. *IEEE Transactions on Computers*, 63(8):2011–2024, May 2014.

[32] Xiaozhou Li and Michael J Freedman. Scaling IP multicast on datacenter topologies. In *Proc. of ACM CoNEXT'13*, pages 61–72, Santa Barbara, CA, December 2013.

[33] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is more: trading a little bandwidth for ultra-low latency in the data center. In *Proc. of USENIX NSDI'12*, pages 253–266, San Jose, CA, April 2012.

[34] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P. Brighten Godfrey. Jellyfish: Networking data centers randomly. In *Proc. of USENIX NSDI'12*, pages 225–238, San Jose, CA, April 2012.

[35] Dan Li, Henggang Cui, Yan Hu, Yong Xia, and Xin Wang. Scalable data center multicast using multi-class bloom filter. In *Proc. of IEEE ICNP'11*, pages 266–275, Vancouver, Canada, October 2011.

[36] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I. Jordan, and Ion Stoica. Managing data transfers in computer clusters with orchestra. In *Proc. of ACM SIGCOMM'11*, page 98–109, Toronto, Canada, August 2011.

[37] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: a distributed messaging system for log processing. In *Proc. of ACM Workshop on Networking Meets Databases (NetDB'11)*, pages 1–7, Athens, Greece, June 2011.

[38] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In *Proc. of ACM IMC'10*, pages 267–280, Melbourne, Australia, November 2010.

[39] Parisa Jalili Marandi, Marco Primi, Nicolas Schiper, and Fernando Pedone. Ring paxos: A high-throughput atomic broadcast protocol. In *Proc. of IEEE/IFIP DSN'10*, pages 527–536, Chicago, IL, June 2010.

[40] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proc. of USENIX HotCloud'10*, pages 1–7, Boston, MA, June 2010.

[41] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Understanding data center traffic characteristics. *ACM SIGCOMM Computer Communication Review*, 40(1):92–99, 2010.

[42] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: A scalable and flexible data center network. In *Proc. of ACM SIGCOMM'09*, page 51–62, Barcelona, Spain, October 2009.

[43] Petri Jokela, András Zahemszky, Christian Esteve Rothenberg, Somaya Arianfar, and Pekka Nikander. Lipsin: Line speed publish/subscribe inter-networking. In *Proc. of ACM SIGCOMM'09*, pages 195–206, Barcelona, Spain, August 2009.

[44] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[45] Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[46] Matthew L Massie, Brent N Chun, and David E Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004.

[47] Dejan Kostić, Adolfo Rodriguez, Jeannie Albrecht, and Amin Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *Proc. of ACM SOSP'03*, pages 282–297, Bolton Landing, NY, October 2003.

[48] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003.

[49] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. Splitstream: high-bandwidth multicast in cooperative environments. *ACM SIGOPS Operating Systems Review*, 37(5):298–313, 2003.

[50] Bradley Cain, Steve E. Deering, Bill Fenner, Isidor Kouvelas, and Ajit Thyagarajan. Internet Group Management Protocol, Version 3. RFC 3376.

[51] Suman Banerjee, Bobby Bhattacharjee, and Christopher Kommareddy. Scalable application layer multicast. In *Proc. of ACM SIGCOMM'02*, pages 205–217, Pittsburgh, PA, August 2002.

[52] Christophe Diot, Brian Neil Levine, Bryan Lyles, Hassan Kassem, and Doug Balensiefen. Deployment issues for the IP multicast service and architecture. *IEEE Network*, 14(1):78–88, January 2000.

[53] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[54] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[55] Multicast Command Reference for Cisco ASR 9000 Series Routers. https://bit.ly/3AaVGDQ. [Online; accessed February 2022].

[56] NetFPGA SUME Reference Learning Switch Lite. https://bit.ly/2UrUFlx. [Online; accessed February 2022].

[57] 10Gb Ethernet: The Foundation for Low-Latency, Real-Time Financial Services Applications and Other, Latency-Sensitive Applications. https://bit.ly/33xtYST. [Online; accessed February 2022].

[58] Apache ActiveMQ. http://activemq.apache.org. [Online; accessed February 2022].

[59] Apache Hadoop. https://hadoop.apache.org/. [Online; accessed February 2022].

[60] Apache openwhisk. https://openwhisk.apache.org/. [Online; accessed February 2022].

[61] Bess (berkeley extensible software switch). https://github.com/NetSys/bess. [Online; accessed February 2022].

[62] Bringing Multicast to the Cloud. https://bit.ly/3jP7naY. [Online; accessed February 2022].

[63] Data plan development kit (DPDK). https://intel.ly/2GYxLAV. [Online; accessed February 2022].

[64] gRPC - An RPC library and framework. https://github.com/grpc/grpc. [Online; accessed February 2022].

[65] Introducing data center fabric, the next-generation Facebook data center network. https://bit.ly/3bWEDKG. [Online; accessed February 2022].

[66] Multicast group capacity: Extreme comes out on top. https://bit.ly/2H5sQ1n. [Online; accessed February 2022].

[67] Open Config, Streaming Telemetry. https://bit.ly/3kf7EEj. [Online; accessed February 2022].

[68] RabbitMQ. http://www.rabbitmq.com. [Online; accessed February 2022].

[69] Using reliable multicast for data distribution with opendds. https://bit.ly/3bWFefo. [Online; accessed February 2022].

[70] Why the world's largest hadoop installation may soon become the norm. https://tek.io/33gDCsU. [Online; accessed February 2022].

[71] Xiaoye Sun. LDA Data Generator. https://github.com/sunxiaoye0116/data_generator/tree/dev. [Online; accessed February 2022].

Appendix A   Supplementary Materials

This appendix includes materials that complement the contents presented in the paper.

A.1   Encoding Spine Downstream Links

The pseudo code of the ENCSPINEDSLINKS algorithm is shown in Algorithm 2. This algorithm encodes spine downstream links of the multicast tree into a label D and calculates the state $\mathbb{S}$ to be maintained by spine switches.

A.2   Processing Spine Downstream Labels

The pseudo code of the PROCSPINEDSLABEL algorithm is shown in Algorithm 3. The algorithm processes two spine downstream labels: the common links among spine switches in the tree (denoted by $\mathbb{C}$), and the filter that encodes the remaining spine downstream links (denoted by D).

A.3   Overheads of Orca

Multicast offers significant bandwidth savings compared to unicast, and thus, it can scale data-intensive tasks that dominate datacenter networks. The authors of [36] reported that the communication time of data-intensive tasks using unicast can be larger than the computation time, especially as the number of workers increases. Achieving the benefits of multicast has been a long-standing problem. Orca achieves the benefits of multicast at the expense of the small overheads described below.

Server Resources. Orca agents require processing resources at servers. However, the computation performed on packets (mostly replacing labels) is quite simple and the memory needed to store leaf labels is small. Thus, Orca agents can easily be implemented on SmartNICs, which are getting popular in datacenters [3]. In this case, no CPU cores are taken away from the servers. Orca agents can also run on regular CPU cores. In this case, the agents consume only a small fraction of the available computing resources in each rack, as shown in the evaluation section. We note that since Orca is a multicast paradigm, the sender in the session transmits only one copy of each packet regardless of the number of receivers in the session. In contract, in unicast, the sender needs to send a separate copy of each packet to every receiver, which for large-scale applications with many receivers and/or high data rates requires allocating additional CPU cores at the sender to sustain the needed data rate. That is, at the whole system level, the CPU resources used by Orca agents can be offset by the savings of CPU resources at the sender.

Packet Latency. Orca adds latency to packets at the leaf layer only, because the packets need to be sent to Orca agents for relabeling. This latency is in the order of one RTT within the rack, because of the simple processing done on packets by

---

Algorithm 2 Encode spine downstream links.

Input: $\mathbb{T}$: multicast tree
Input: $\mathbb{C}$: common ports in spine downstream switches
Input: F: filter size in bits
Output: D: computed spine downstream label
Output: $\mathbb{S}$: state sent to a subset of the spine switches

```
 1: function ENCSPINEDSLINKS(𝕋, ℂ, F)
 2:    Ⓐ Calculate a spine downstream label
 3:    D = BitString(size=F)
 4:    for (l ∈ 𝕋.spine_ds_links()) do
 5:        if (l ∉ ℂ) then
 6:            D = D ∪ encode(l.id)
 7:    Ⓑ Calculate false positive candidates
 8:    cands = { }
 9:    for (u ∈ 𝕋.spine_switches()) do
10:        for (l ∈ u.ds_links()) do
11:            if (l ∉ 𝕋.spine_ds_links()) then
12:                cands = cands ∪ (u, l.dst)
13:    Ⓒ Calculate spine switch state
14:    𝕊 = { }
15:    for (l ∈ cands) do
16:        if (check(l.id, D) and l ∉ ℂ) then //false positive
17:            𝕊 = 𝕊 ∪ {⟨l.src, l.id⟩} // add link to state
18:    return ⟨D, 𝕊⟩
```

Orca agents. Most throughput-intensive datacenter applications, e.g., MapReduce [44], Hadoop [59], and Spark [40], can easily tolerate this small latency [33].

Communications Overheads. Orca achieves substantial bandwidth savings compared to the commonly-used unicast model. Orca, on the other hand, attaches a small, fixed-sized label (19 bytes) to each packet in typical datacenters; Orca label is smaller than the IP header. In addition, Orca uses additional bandwidth between leaf switches and agents deployed on servers within the same rack. Prior studies, however, reported that links at leaf layer are under-utilized. For instance, the study in [38] has found that the datacenter edge is lightly utilized: 80% of the time, the utilization is less than 10% for cloud and enterprise datacenters. A recent study by Facebook [20] reported that links between leaf switches and servers have a 1-minute average utilization of less than 1%.

A.4   Extensions and Limitations of Orca

Multipath Routing. In Orca, the multicast tree has one path from the source VM to any core switch, then it reaches the receivers by branching to spine and leaf switches. Orca can support multipath routing to achieve reliability and load balancing as follows. The centralized controller can compute multiple trees, each has the same source and receivers of the session but consists of different links. For example, the centralized controller can choose a different core switch as the root for each tree. It then calculates a different source label

---

Algorithm 3 Process a spine downstream label.

Input: D: spine downstream label
Input: l: downstream link attached to the spine switch
Input: $\mathbb{C}$: set of common ports in spine downstream switches
Input: $State$: state maintained at the spine switch
Output: **true** if duplicating a pkt on link l, else **false**

    // Runs for every link attached to the spine switch
1: **function** PROCSPINEDSLABEL(D, l, $State$)
      // Links belonging to $\mathbb{C}$
2:     **If** $index(l.id) \in \mathbb{C}$ **then return true**
      // Checking the filter for $index(l.id) \notin \mathbb{C}$
3:     **If not** $check(l.id, D)$ **then return false**
      // sID is the session ID included in the packet header
4:     **return** $l.id \notin State[sID]$

---

for each tree, and instructs the source VM to store the new labels and spine switches to maintain state (if needed). The source attaches different source labels to the packets to instruct switches to forward them on links of different trees. Leaf labels are identical for all trees as they have the same receivers. As in other multipath routing systems, packet reordering may occur in this case and would need to be handled by the application.

Reliability and Congestion Control in Multicast. Prior works, e.g., [9, 31], proposed various methods for reliable transmission and congestion control for datacenter multicast. These methods can be used on top of Orca. In addition, the Orca agent can reduce the number of control messages, e.g., ACK or NACK, since it can aggregate them per rack.

Incremental Deployment. Orca can run on legacy switches by encapsulating its labels in VLAN or VXLAN headers. The header identifier can be used to instruct switches to duplicate incoming packets.

Limitations of Orca. Deploying Orca in graph-based datacenter networks, e.g., Jellyfish [34] and Xpander [13], may require changes in some components of Orca. For example, although our server-assisted approach will work at the leaf layer in Jellyfish, Jellyfish's lack of structure does not allow Orca to use the same algorithms at other layers. A new label calculation algorithm would need to be designed to encode tree links without imposing assumptions on their layers.

A.5 Additional Simulation Results

We evaluate the running time of Orca and Elmo spent by the centralized controller when sessions change.

Running Time. Orca is simple and enables more updates per second to be processed by the control plane. For the Uni-S dataset, the average running time for Orca to calculate a session label is 0.34 ms (SD is 0.4 ms), while this average is up to 7.286 ms (SD is 9.8 ms) for Elmo-3. The average (SD) running times for Elmo-1, Elmo-2 and Elmo-4 are 6.1 ms (5.7 ms), 5.5 ms (5.6 ms) and 6.5 ms (8.6 ms), respectively. These times were measured on a workstation with a 2.3 GHz CPU.

Summary: Orca calculates labels 21X faster than Elmo.

Yeti: Stateless and Generalized Multicast Forwarding

Khaled Diab   Mohamed Hefeeda

School of Computing Science
Simon Fraser University
Burnaby, BC, Canada

Abstract

Current multicast forwarding systems suffer from large state requirements at routers and high communication overheads. In addition, these systems do not support generalized multicast forwarding, where traffic needs to pass through traffic-engineered paths or requires service chaining. We propose a new system, called Yeti, to efficiently implement generalized multicast forwarding inside ISP networks and supports various forwarding requirements. Yeti completely eliminates the state at routers. Yeti consists of two components: centralized controller and packet processing algorithm. We propose an algorithm for the controller to create labels that represent generalized multicast graphs. The controller instructs an ingress router to attach the created labels to packets in the multicast session. We propose an efficient packet processing algorithm at routers to process labels of incoming packets and forwards them accordingly. We prove the correctness and efficiency of Yeti. In addition, we assess the performance of Yeti in a hardware testbed and using simulations. Our experimental results show that Yeti can efficiently support high speed links. Furthermore, we compare Yeti using real ISP topologies in simulations against the closest systems in the literature: a rule-based approach (built on top of OpenFlow) and two label-based systems. Our simulation results show substantial improvements compared to these systems. For example, Yeti reduces the label overhead by 65.3%, on average, compared to the closest label-based multicast approach in the literature.

1 Introduction

Recent large-scale Internet applications have introduced a renewed interest in scalable multicast services. Examples of such applications include live Internet broadcast (e.g., Facebook Live), IPTV [27], webinars and video conferencing [22], and massive multiplayer games [26]. The scale of these applications is unprecedented. For instance, due to the COVID-19 pandemic, a recent study [2] reported an increase by one order of magnitude within two months in video conferencing traffic passing though a major European ISP. Moreover, Facebook Live aims to stream millions of live sessions to millions of concurrent users [8, 41]. To reduce the network load of such applications, ISPs can use multicast to efficiently carry the traffic through their networks. Examples of commercial systems using multicast include AT&T UVerse [40] and BT YouView [37]. Beyond multimedia systems, multicast is also useful for various applications such as real-time stock market updates, cloud applications [33], and publish-subscribe systems [24, 36, 39]. For instance, the CIO of the Japan Exchange Group highlighted the importance of multicast for their stock trading operations [32].

Large ISPs need to support *generalized* multicast forwarding to handle various business requirements. Specifically, providers of large-scale live applications require ISPs carrying their traffic to meet target quality metrics or SLAs (service level agreements), especially for popular/paid live multicast sessions. To meet the SLAs for various customers, ISPs may need to direct the traffic to network paths different from the minimum-cost ones computed by the routing protocols deployed in the ISP network. This is usually referred to as *traffic engineering*. Prior works, e.g., [7, 11, 12, 16], have proposed algorithms to support various traffic engineering objectives.

In addition, ISP customers may require their multicast traffic to pass though an *ordered* sequence of network services such as firewall, intrusion detection, and video transcoding before reaching the destinations. This is referred to as *service chaining*. Network services are usually deployed as virtual functions running on servers attached to *some* of the core routers in the ISP network. Previous works, e.g., [1, 5], presented algorithms for calculating optimal network paths to satisfy service chaining requirements.

Given the service chaining and traffic engineering requirements of recent applications, multicast sessions can no longer be represented as simple spanning trees. Rather, they need to be represented as general graphs. Efficiently forwarding traffic of multicast sessions represented as arbitrary graphs is, however, a challenging problem. One of the main concerns is the *state* that needs to be maintained at routers, which grows

linearly with the number of multicast sessions. This state also needs to be frequently updated to handle session changes and network dynamics, which imposes substantial communication and processing overheads, especially on core routers that need to support high-speed links carrying numerous sessions.

In this paper, we address the lack of scalable and generalized multicast forwarding systems for large-scale ISPs. In particular, we propose a *fully stateless* approach, called Yeti, to implement generalized multicast graphs. Yeti supports fast adaptation to network dynamics such as routers joining/leaving sessions and link failures, and it does not impose significant communication overheads. To the best our knowledge, Yeti is the first multicast forwarding system that supports multicast sessions with traffic engineering and service chaining requirements. A high-level overview of Yeti is illustrated in Figure 1.

The main idea of Yeti is to completely move the forwarding information for each graph to the packets themselves as labels. Designing and processing such labels, however, pose key challenges that need to be addressed. First, we need to efficiently encode the graph forwarding information in as few labels as possible. Second, the processing overheads and hardware usage at routers need to be minimized. This is to support many concurrent multicast sessions, and to ensure the scalability of the data plane. Third, forwarding packets should not introduce *ambiguity* at routers. That is, while minimizing label redundancy and overheads, we must guarantee that routers will forward packets *on and only on* the links of the multicast graph. Yeti addresses these challenges.

This paper makes the following contributions:

- We propose a generalized multicast forwarding system that completely eliminates the state at routers; a long-standing problem for multicast. The proposed system supports service chaining and traffic engineering requirements.
- We design a control-plane algorithm to calculate an optimized label for each generalized multicast graph.
- We design an efficient packet processing algorithm for routers to handle labels attached to packets. The algorithm forwards packets only on links of the multicast graph. And it does not introduce any redundant traffic or create loops in the network.
- We present proofs to show the correctness of Yeti.
- We implement Yeti in a hardware testbed using a programmable router (NetFPGA) to demonstrate its practicality. Our results show that Yeti can support high-speed links carrying thousands of multicast sessions.
- We compare Yeti against a rule-based approach implemented using OpenFlow and the closest label-based approaches, LIPSIN [25] and BIER-TE [3], in simulations using real ISP topologies with different sizes. Our simulation results show that unlike Yeti which does not maintain state at any core router, the rule-based approach requires maintaining state at every router in the session
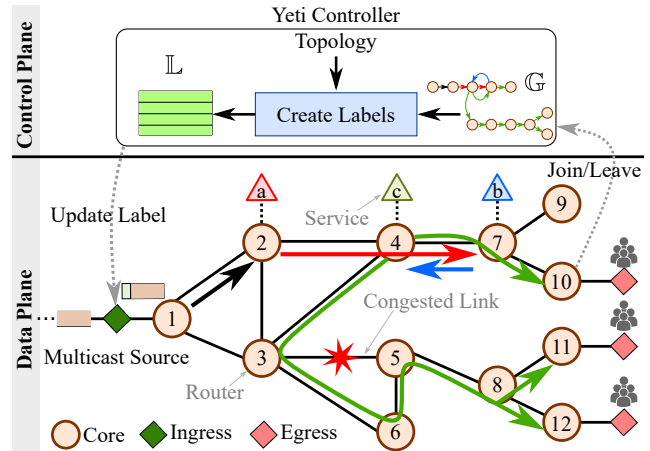

Figure 1: High-level overview of Yeti.

and LIPSIN maintains state at about 20% of the routers. In addition, Yeti reduces the label overhead by 65.3%, on average, compared to BIER-TE.

2   Related Work

We divide the related multicast forwarding works in the literature into stateful, stateless, and hybrid approaches.

Stateful Multicast Approaches. These multicast approaches require storing forwarding state about sessions at routers. The traditional IP multicast [31] is an example of such approaches. IP multicast is implemented in core routers produced by most vendors. However, it suffers from scalability issues in real deployments [29]. In particular, the group management and tree construction protocols, e.g., IGMP [28] and PIM [13], require maintaining state at routers for each session, and they generate control messages among routers to refresh and update this state. Thus, in practice, router manufacturers tend to limit the number of multicast sessions [38]. In addition, IP multicast uses shortest paths and cannot implement generalized graphs.

Recent SDN-based protocols, e.g., OpenFlow [17], can implement rule-based approaches, where a controller installs header-matching rules and actions to forward/duplicate packets. Since OpenFlow stores state at every router along the multicast trees, the total forwarding state at routers grows with the number of sessions.

Stateless Multicast Approaches. There are a few recent proposals for designing stateless multicast forwarding systems. For example, BIER [10] encodes global router IDs of tree receivers in the label as a bitmap. BIER supports only shortest paths and cannot realize traffic-engineered ones. A recent amendment to BIER, called BIER-TE [3], supports traffic-engineered trees. BIER-TE maps each bit position in the label to one of the links attached to routers in the network. It encodes the links of a multicast tree as corresponding bit positions in the calculated label. Upon receiving a packet,

a BIER-TE router checks the bit positions in the label. If the router matches one of its links in the label, it clears its position in the label and forwards/duplicates the packet on that link. The bitmap structure used in BIER-TE allows it to only implement multicast sessions represented as trees, and it cannot implement general multicast distribution graphs, as such graphs could have cycles to allow the multicast traffic to be processed by the specified set of network services. This is illustrated in the example multicast distribution graph in Figure 1, where packets of the session traverse the link between routers 4 and 7 three times to be processed by the services a→b→c.

Hybrid Multicast Approaches. Yeti is not the first system that moves forwarding information as labels attached to packets. However, prior systems did not support generalized forwarding, and they needed to maintain state at some or all routers belonging to the multicast tree. We refer to these systems as hybrid approaches. For example, the early work by Chen et al. [30] proposed a label-based system that attaches link IDs to every packet in a multicast session, and removes unwanted portions of the label as the packet traverses the network. The processing algorithm in [30] requires maintaining state at every router belonging to a multicast session in order to remove the labels, which is not scalable. Later works, e.g., mLDP [23], enable multicast in label-switched paths (LSPs). mLDP forwards traffic on the shortest paths and thus cannot support traffic-engineered trees. It also requires an additional protocol to distribute labels among routers.

LIPSIN [25] uses a Bloom filter as label to encode global link IDs of a tree. LIPSIN may result in redundant traffic or forwarding loops, because of the probabilistic nature of Bloom filters. Thus, LIPSIN requires an additional protocol where downstream routers notify upstream ones if they falsely receive a packet. This protocol imposes additional state and communication overheads on routers.

Segment routing (SR) [4] is a recent proposal to support traffic-engineered unicast flows. It was later extended to support multicast by considering every tree as a segment in the network. It attaches one label containing the tree ID to packets of a session. Routers along the tree maintain a mapping between that label and the output interfaces. That is, the SR multicast extensions require maintaining state at routers for every tree.

In §5, we compare Yeti versus a rule-based approach implemented in OpenFlow, LIPSIN, and BIER-TE as they are the closest stateful, hybrid, and stateless multicast systems, respectively, that can support traffic engineering requirements.

3 Problem Definition and Solution

We start this section by specifying the considered problem and its challenges. We next describe an overview of Yeti and its main components. This is followed by the details of each component. In the **Appendix** §C, we present an illustrative example of Yeti to demonstrate its details.

3.1 Problem Definition and Challenges

The problem considered in this paper is how to efficiently forward the traffic of a *generalized* multicast session that may need to be processed by an ordered set of network services and/or directed through a specific set of network paths within the ISP network.

For illustration, consider the ISP network in the lower part of Figure 1, which contains ingress, core, and egress routers marked by different shapes and colors. Some of the core routers are connected to servers offering various (virtualized) network services such as intrusion detection and video transcoding. There is a multicast source connected to the ingress router and multiple receivers reachable through the three egress routers. The creator of the multicast session requires the traffic to be processed by the three network services a→b→c in order. In addition, the ISP implements traffic engineering mechanisms for various objectives, e.g., to minimize the maximum link utilization (MLU), which requires the traffic to avoid the link $3 \rightarrow 5$ in this example. The colored arrows in the figure show the different paths taken by the traffic of the multicast session to reach all receivers. These paths form a graph (not tree), which we refer to as the *multicast distribution graph* $\mathbb{G}$. Note that nodes in the distribution graph represent routers, not end users. The top right part of Figure 1 shows the graph for the multicast session marked in the lower part of the figure.

Our problem then becomes how to get routers in the ISP to forward the traffic of a multicast session represented by an arbitrary multicast distribution graph $\mathbb{G}$. Existing algorithms in the literature, e.g., [1, 12], can be used to calculate $\mathbb{G}$ to satisfy various service chaining and traffic engineering requirements; our proposed (forwarding) solution is orthogonal to the calculation of $\mathbb{G}$ and can work with any of them.

The arbitrary nature of the distribution graph makes designing scalable multicast forwarding systems a challenging problem. A possible approach to address this problem is to maintain state (e.g., in the form of match-action rules) at routers. However, as mentioned in §2, maintaining state at routers is not scalable even for traditional multicast forwarding without service chaining and traffic engineering requirements. This is because the state, which grows linearly with the number of multicast sessions, not only consumes the scarce SRAM resources of routers, but it also needs to be frequently updated to handle network failures and session dynamics (e.g., router joining/leaving). The cost of updating the state consists of two factors. First, routers need to process many update (control) messages while processing data packets, which may result in slowing down the data plane operations. Second, frequent state updates negatively impact the network agility and consistency, because the control plane has to *schedule* the

updates to corresponding routers to ensure consistency [20], since greedy state updates may result in violating the SLA objectives [6].

To reduce state, a part or all of the forwarding information can be moved to labels which are attached to the packets of multicast sessions, where routers use these labels in the forwarding decisions. However, efficiently representing multicast graphs in compact labels is difficult, especially for multicast sessions that have service chaining and traffic engineering requirements. If not carefully designed, labels representing a multicast graph can grow large in size and hence impose significant communication overheads, and more critically they could introduce *ambiguity* at routers, i.e., routers may not be able to decide which interface(s) to forward the packets on. This may introduce duplicate packets and forwarding loops, which substantially increases the load on the ISP network and wastes its bandwidth and processing resources.

In summary, forwarding generalized multicast graphs presents multiple challenges that Yeti addresses. Specifically, stateful approaches *reduce scalability* as they impose substantial memory and processing overheads on switches. On the hand, current stateless approaches significantly *increases packet sizes* and may *introduce forwarding ambiguity*. This would defeat the main purpose of deploying multicast in the first place. Yeti breaks this long-standing trade-off between scalability, efficiency, and correctness by completely moving the forwarding state into compact labels, and carefully processing them in the data plane.

3.2 Solution Overview

Yeti is a *stateless* multicast forwarding system that efficiently implements general multicast graphs inside a single ISP network; extending Yeti to support multiple ISPs is described in §3.6. As shown in Figure 1, the ISP network has data and control planes. The data plane is composed of routers. Every router is assigned a unique ID, and it maintains two data structures: Forwarding Information Base (FIB) and interface list. FIB provides reachability information between routers using shortest paths. The interface list maintains the IDs of all local interfaces. The control plane (or the *controller*) learns the ISP topology, shortest paths between routers, and interface IDs for every router, which is done using common intra-domain routing and monitoring protocols.

Yeti consists of a centralized controller and a packet processing algorithm. The controller calculates the distribution graph for a multicast session using existing algorithms, e.g., [1, 12], and it creates, using our algorithm in §3.4, an optimized set of labels $\mathbb{L}$ to realize this graph in the data plane. As detailed in §3.3, Yeti defines four types of labels; each serves a specific purpose in encoding the graph efficiently. The controller sends the set of labels $\mathbb{L}$ to the ingress router of the session, which in turn attaches them to all packets of the session. When a graph $\mathbb{G}$ changes (due to link failure

Name	Type	Content	Content Size (bits)
FSP	00	Global router ID	$1 + \lceil \log_2 N \rceil$
FTE	01	Local interface ID	$\lceil \log_2 I \rceil$
MCT	10	Interface bitmap	I
CPY	11	Label range (in bits)	$\lceil \log_2 (N \times \text{size(FTE)}) \rceil$

Table 1: Label types in Yeti. N is the number of routers, and I is the maximum number of interfaces per router.

or egress router joining/leaving the session), the controller creates a new set of labels and sends them *only* to the ingress router, no other routers need to be updated.

The packet processing algorithm, described in §3.5, is deployed at core routers. It processes the labels attached to packets and forwards/duplicates packets based on these labels. It also determines the subset of labels to attach to the forwarded packets such that the subsequent routers can realize the distribution graph without any ambiguity, forwarding loops, or redundant traffic. We present a theorem proving the correctness of Yeti in §3.6.

We present an illustrative example in the **Appendix**. This example implements the distribution tree in Figure 1, and it shows the details of creating labels at the controller and processing packets at routers.

3.3 Label Types in Yeti

Yeti is a label-based system. Thus, one of the most important issues is to define the types and structures of labels in order to minimize the communication and processing overheads, while still being able to represent generalized multicast graphs. We propose the four label types shown in Table 1. The first two label types are called *Forward Shortest Path* (FSP) and *Forward Traffic Engineered* (FTE). They are used by routers to forward packets on paths that have no branches. The other two label types are *Multicast* (MCT) and *Copy* (CPY). The MCT label instructs routers to duplicate a packet on multiple interfaces, and the CPY label copies a subset of the labels. Every label consists of two fields: *type* and *content*. The *type* field is used by routers to distinguish between labels during parsing, and the *content* field contains the information that this label carries. The size of a label depends on the size of the *content* field.

We use the example topology in Figure 2 to illustrate the rationale used in defining Yeti labels. In the figure, solid lines denote tree links and the dotted line denotes a link on the shortest path to some destinations. The ISP avoids it because it is congested in this example.

We divide a multicast graph into *path segments* and *branching points*. A path segment is a contiguous sequence of routers without branches. If a path satisfies any sub-sequence of the service chaining requirements of a session, the path segment
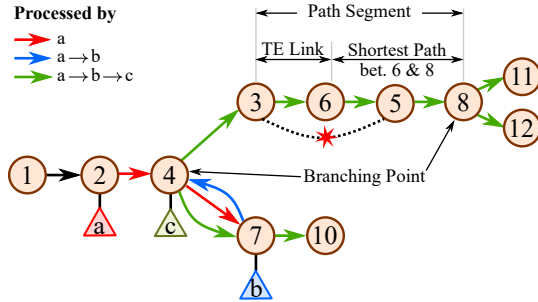
Figure 2: Illustration of path segments and branching points in Yeti. Segment $3 \rightarrow 8$ does not follow the shortest path.

Figure 3: The resulting tree $\mathbb{T}$ for the graph $\mathbb{G}$ in Figure 2.

ends when there is a router with at least one service. A branching point refers to a router that duplicates packets on multiple interfaces. For the example in Figure 2, there are five path segments: $\{1 \rightarrow 2\}$, $\{2 \rightarrow 4 \rightarrow 7\}$, $\{7 \rightarrow 4\}$, $\{7 \rightarrow 10\}$, and $\{3 \rightarrow 6 \rightarrow 5 \rightarrow 8\}$. Routers 4 and 8 are branching points.

The basic label in Yeti is FTE, where a router is instructed to forward the packet carrying the label on a specific interface. In many situations, however, packets follow a sequence of routers on the shortest path. For these situations, we define the FSP label, which replaces multiple FTE labels with just one FSP label. An FSP label contains a global router ID, which instructs routers to forward incoming packets on the shortest path to that router. In addition, the first bit in an FSP label indicates whether the packet will be processed at the corresponding router. For example, in Figure 2, instead of using two FTE labels for the links $\{6 \rightarrow 5\}$ and $\{5 \rightarrow 8\}$, Yeti uses one FSP label with destination ID set to node 8. In large topologies, FSP labels significantly reduces label overheads.

FTE and FSP labels can represent path segments, but they cannot handle branching points where packets need to be duplicated on multiple interfaces. Notice that, after a branching point, each branch needs a different set of labels because packets will traverse different routers. To handle branching points, we introduce the MCT and CPY labels. The MCT label instructs routers to duplicate packets on multiple interfaces using a bitmap of size I bits, where I is the maximum interface count per router. The bitmap represents local interface IDs, where the bit locations of the interfaces that the packet should be forwarded on are set to one. The CPY label does not represent a forwarding rule. Instead, it instructs a router to copy a subset of labels when duplicating packets to a branch without copying all labels. Specifically, consider a router that duplicates packets to n branches. The MCT label is followed by n sets of labels to steer traffic in these branches, where every label set starts with a CPY label. The CPY label of one branch contains an offset of label sizes (in bits) to be duplicated to that branch. For example, in Figure 2, router 4 will process an MCT label followed by two CPY labels for the traffic represented with a green arrow, one for each of the two branches. The CPY label content size in Table 1
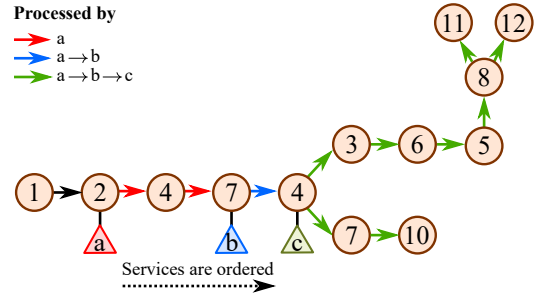
uses the worst-case scenario. This happens when the graph has the largest diameter, which is $O(N)$ and every link is traffic-engineered, where N is the number of core routers.

3.4 Creating Yeti Labels at the Controller

The ENCODEGRAPH algorithm, shown in Algorithm 1, runs at the controller to create labels. We omit the pseudo code for some functions that the algorithm calls due to space limitations. When the distribution graph $\mathbb{G}$ changes, the algorithm calls the BUILDTREE function to create a tree $\mathbb{T}$ that reflects the order of network services needed before reaching the destinations. Then, the ENCODEGRAPH algorithm calls the CREATELABELS function with the created tree to calculate a new set of labels $\mathbb{L}$ to encode the graph paths and sends them to the ingress router, which attaches them to every packet in the session. The details of our algorithms are as follow.

Building the Tree. The BUILDTREE function traverses the multicast graph and creates a list of tuples for every path and provided services on that path. For example, in Figure 2, the tuple $\langle \{7 \rightarrow 4\}, \{a \rightarrow b\} \rangle$ represents packets traversing the path $\{7 \rightarrow 4\}$ after processed by the services a and b.

The BUILDTREE function then traverses these tuples from the tuple starting with the source node. The function keeps track of the current parent and visited nodes in the tree, and builds the tree $\mathbb{T}$ as follows. For every tuple, the function creates nodes with every router ID and the provided services in that tuple. For the tuple mentioned earlier, the function creates the nodes: $(7, \{a \rightarrow b\})$ and $(4, \{a \rightarrow b\})$. The function adds a node to $\mathbb{T}$ if it did not exist before. Then, the function adds that node to the children of current parent, and sets the new node as the current parent. If a node exists in the tree, the function sets it as the current parent. Figure 3 depicts the resulting tree of the graph in Figure 2.

Creating Labels. The CREATELABELS algorithm divides $\mathbb{T}$ into segments and branching points. The algorithm calculates FSP and FTE labels for every segment, and MCT and CPY labels at branching points. The label order is important because it reflects which routers process what labels. The algorithm traverses the core routers of $\mathbb{T}$ in a depth-first search order starting from the core router connected to the ingress router.

Algorithm 1 Encode a multicast graph into labels.

Input: $\mathbb{G}$: multicast graph
Input: S: ordered list of services in the session
Input: $\mathbb{P}$: shortest path map
Output: $\mathbb{L}$: labels to be sent to the ingress router

1: **function** ENCODEGRAPH($\mathbb{G}$, S, $\mathbb{P}$)
2:     $\mathbb{T}$ = BUILDTREE($\mathbb{G}$, S)
3:     **return** CREATELABELS($\mathbb{G}.src$, $\mathbb{T}$, S, $\mathbb{P}$)
4: **function** CREATELABELS($source$, $\mathbb{T}$, S, $\mathbb{P}$)
5:     $\mathbb{L} = \{\}$, $pth\_seg = \{\}$, $stack = \{source\}$
6:     **while** ($stack$.size() > 0) **do**
7:         $r = stack$.pop() // a router in $\mathbb{T}$
8:         // Services provided by r for the session
9:         $srv = S$.at(r)
10:        $core\_children = \mathbb{T}$.core_children($r$)//core routers
11:       $children = \mathbb{T}$.children(r) //core and egress routers
12:       // Build a path segment pth_seg
13:       **if** ($core\_children$.size() $== 1$) **then**
14:         pth_seg.append(r); $stack$.push($children[0]$)
15:       // Handle a path segment (create FSP & FTE)
16:       // $S[0]$ is the next expected service
17:       **if** ($core\_children$.size() $== 0$ **or** $S[0] \in srv$) **then**
18:         pth_seg.append(r)
19:         $lbls$ = CALCSEGMENTLBL($\mathbb{T}$, pth_seg, $\mathbb{P}$)
20:         $\mathbb{L}$.push($lbls$); $pth\_seg = \{\}$
21:         S.remove(srv)
22:       // Handle branching point (create MCT & CPY)
23:       **else if** ($children$.size() > 1) **then**
24:         **if** (pth_seg.size() > 0) **then**
25:             pth_seg.append(r)
26:             $lbls$=CALCSEGMENTLBL($\mathbb{T}$, pth_seg, $\mathbb{P}$)
27:             $\mathbb{L}$.push($lbls$); $pth\_seg = \{\}$
28:         $\langle mct\_lbl, cpy \rangle$ = CREATEMCT($children$)
29:         $\mathbb{L}$.push(mct_lbl)
30:         **if** (cpy) **then** // Creating CPY labels
31:             **for** ($c \in children$) **do**
32:                 // A recursive call for each branch
33:                 br_lbls = CREATELABELS(c, $\mathbb{T}$, S, $\mathbb{P}$)
34:                 $offset$ = CALCLABELSIZE(br_lbls)
35:                 $\mathbb{L}$.push(CPY($offset$));$\mathbb{L}$.push(br_lbls)
36:     **return** $\mathbb{L}$

It keeps track of the router r that is being visited, and one path segment (pth_seg). Once a router r is visited, if r has only one core child (Line 13 in Algorithm 1), this means that r belongs to the current segment. The algorithm then appends r to pth_seg, and pushes its child to the stack to be traversed later. For example, the algorithm pushes routers 3, 6, 5 and 8 in Figure 3 to pth_seg because each of them has only one child. If r has no core children or it provides some services (has a path segment), or r has more than one child (has a branching point), the algorithm calculates labels as follows.

Handling Path Segments. The CREATELABELS algorithm creates a label for a path segment when pth_seg ends. This happens in three cases. First, when r is connected to an egress router (e.g., router 10 in Figure 3). Second, when r is a branching point and pth_seg is not empty (e.g., router 8 in Figure 3). Third, when r provides at least on service (e.g., router 2 in Figure 3). In all cases, the algorithm appends r to pth_seg and calculates FSP and FTE labels using CALCSEGMENTLBL.

CALCSEGMENTLBL takes as inputs a tree $\mathbb{T}$, a path segment pth_seg and the shortest path map $\mathbb{P}$, and calculates the FSP and FTE labels of the given pth_seg. It divides pth_seg into two sub-paths: one that follows the shortest path, and one that does not. It then recursively applies the same to the latter sub-path. Specifically, CALCSEGMENTLBL concurrently traverses pth_seg and the shortest path between source and destination. It stops when the traversal reaches a router in pth_seg that does not exist in the shortest path. This means that this router does not follow the shortest path, hence, it adds an FSP label for the previous router. If pth_seg has routers that do not follow the shortest path, CALCSEGMENTLBL adds an FTE label and recursively calls itself using a subset of pth_seg that is not traversed so far. CALCSEGMENTLBL does not generate two consecutive FSP labels. When it calculates an FSP label, it either terminates, or creates an FTE label followed by a recursive call.

For the example in Figure 3, the CALCSEGMENTLBL algorithm processes the segment $\{3 \rightarrow 6 \rightarrow 5 \rightarrow 8\}$ as follows. It finds that the link $(3, 6)$ is not on the shortest path from 3 to 8. It calculates an FTE label for this link, and recursively calls itself with the sub-path $\{6 \rightarrow 5 \rightarrow 8\}$ as an input, for which the algorithm creates an FSP label with router ID 8.

Handling Branching Points. The CREATELABELS algorithm calculates MCT and CPY labels at branching points. The algorithm calls CREATEMCT that returns MCT label and a boolean value cpy indicating whether CPY labels are required. To create an MCT label, CREATEMCT initializes an empty bitmap of width $I + 1$ (I is the maximum interface count per router). For every child c of r, it sets the bit location in this bitmap that represents the interface ID between r and c. It checks if CPY labels are needed as follows. If any child c has at least one core child, this means that this core child needs labels to forward/duplicate packets. Otherwise, if all children have no other core children, the router r is either directly connected to an egress router, or its children are connected to egress routers. Thus, these routers do not need more labels and Yeti does not create CPY labels for these branches. For example, at router 4 in Figure 3, core children 3 and 7 have other core children which are 6 and 10, respectively. Hence, two CPY labels are created for the two branches at 4. The algorithm does not create CPY labels at router 8, because its core children 11 and 12 have no other core children.

Recall that a CPY label copies a subset of labels at a specific branch. If CPY labels are needed at the branching point and r has n children/branches, the MCT label is followed by

n CPY labels, and every CPY label is followed by labels to forward packets on the corresponding branch. Specifically, the algorithm iterates over the children of *r*. For every child *c*, the algorithm adds an FSP label if the child provides services. Then, the algorithm recursively calls CREATELABELS to create labels of the corresponding branch (Line 33). The created CPY label for a branch contains the size of this branch labels in bits to be copied. We calculate this size by accumulating the size of every label type in *br_lbls* (Line 34).

Time and Space Complexities. In the worst-case scenario, when every router is a branching point, the ENCODEGRAPH algorithm needs to create labels for each branch. Thus, the time complexity of the ENCODEGRAPH algorithm is $O(K^2N^2 + M)$, where N is the number of routers, M is the number of links, and K is the maximum service chain length. We note that the values of N, M and K are not large for realistic ISP networks. The number of ISP routers N is in the range of 10's–100's [9, 18], most ISP networks are sparse with number of links M ranging from 500 to around 2,000, and the length of service chains K ranges from 2–10 [1]. Given these practical values, the ENCODEGRAPH algorithm can easily run on a commodity server. Notice that the CALCSEGMENTLBL algorithm processes the segment after all routers of that segment is traversed. Thus, it only adds linear overhead to the first term of the time complexity. The space complexity of the ENCODEGRAPH algorithm is $O(N^2D)$, where D is the diameter of the network.

3.5 Processing Yeti Packets

The proposed packet processing algorithm is to be deployed at core routers, and it processes Yeti packets. This is done by setting a different Ethernet type for Yeti packets at ingress routers. A core router checks the Ethernet type of incoming packets, and invokes the processing algorithm if they are Yeti packets. The algorithm forwards/duplicates packets and it removes labels that are not needed by next routers. It also copies a subset of labels at branching points.

The packet processing algorithm works as follows. If the packet has no labels, this means the packet reached a core router that is attached to an egress router. So, the packet is forwarded to that egress router. Otherwise, the algorithm processes labels according to the following cases:

(1) FSP Label. If the FSP label content is not the receiving router ID, it means that this router belongs to a path segment. The algorithm then forwards the packet along the shortest path based on the underlying intra-domain routing protocol *without* removing the label. If the FSP content equals the router ID, this means that the path segment ends at this router. The algorithm first checks whether the packet needs to be processed by services connected to that router. If the first bit is set, then the packet is forwarded to the datacenter. Otherwise, the algorithm removes the current label and calls the packet processing algorithm again to process next labels. This is

because the packet may have other labels.
(2) FTE Label. The algorithm removes the label, extracts the local interface ID, and forwards the packet on that interface.
(3) MCT Label. The algorithm first copies the original labels, and removes the labels from the packet. It then extracts the MCT content into *mct*. The MCT label contains the interface ID bitmap (*mct*.intfs) and whether it is followed by CPY labels (*mct*.has_cpy). The algorithm iterates over the router interfaces in ascending order of their IDs. It locates the interfaces to duplicate the packet on. For every interface included in the MCT label, the algorithm clones the packet. If the MCT label is followed by CPY labels, the algorithm removes the corresponding CPY label, extracts the following labels based on the offset value, and forwards the cloned packet on the corresponding interface.

Time and Space Complexities. The time complexity of the algorithm is $O(I)$, where I is the maximum interface count per router. The algorithm does not require additional space at routers.

3.6 Analysis and Practical Considerations

Analysis. The following theorem proves the correctness of Yeti. That is, Yeti forwards packets on and only on links belonging to the multicast graph. This is a critical objective for large-scale multicast sessions, as redundant traffic wastes network resources and overloads routers. Due to space limitation, we show the full proof in the Appendix §A.

Theorem 1 (Correctness). *Yeti forwards packets on and only on links that belong to the multicast graph.*

Proof Sketch. Yeti guarantees correctness by creating an ordered set of labels to realize the given multicast distribution graph. We analyze the order and type of the created labels and prove that they do not result in forwarding loops or redundant traffic while delivering the traffic to all receivers in the multicast session. □

Practical Considerations. The description of Yeti has focused on offering a scalable multicast service within a *single* ISP using programmable routers. In the Appendix §B, we describe simple techniques to enable Yeti across multiple ISPs and its incremental deployment.

4 Evaluation in a Testbed

We present a *proof-of-concept* implementation of the proposed multicast forwarding system, and we conduct experiments in a testbed with a NetFPGA programmable router.

In addition, since switches that support the P4 programming language [21], e.g., Tofino, are getting popular in practice, we also implemented Yeti in P4. We obtained a license for the Intel P4 software development environment (SDE) version 9.5.0, which contains various tools, including a P4
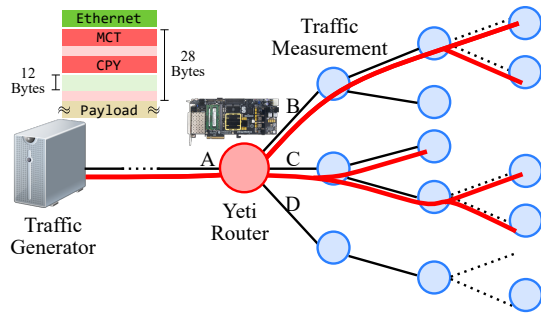
Figure 4: Setup of our testbed.

Latency (μsec)	$50^{th}\%$	$95^{th}\%$	$99.9^{th}\%$
Yeti	960.4	973	1,042
MAC forwarding	960.3	972.9	1,040
Difference	0.1	0.1	2

Table 2: Packet latency (in μsec) measured in our testbed.

compiler (`bf-p4c`) and a switch model. The compiler produces code that runs on Tofino switches. We validated our implementation using the switch model included in the SDE. The details of the P4 implementation of Yeti are presented in Appendix §D.

4.1 Testbed Setup

The testbed, shown in Figure 4, has a Yeti Router representing a core router in an ISP topology that receives and processes packets of concurrent multicast sessions. We implemented the Yeti Router in a programmable processing pipeline using NetFPGA SUME [19], which has four 10GbE ports. The testbed also has a 40-core server with an Intel X520-DA2 2x10GbE NIC, which is used to generate traffic of multicast sessions at high rate using MoonGen [14].

Our router implementation is based on the open source project in [34]. This project contains three main Verilog modules: `input_arbiter`, `output_port_lookup` and `output_queues`. Our implementation modifies the last two modules in the router as follows. The `output_port_lookup` module is modified to read the first label to decide which ports to forward/duplicate the packet on. For each duplicated packet, it decides the labels to be detached and maintains this information in the packet metadata. We also modified the `output_queues` module to runs at every output queue of the router, and detach labels that are not needed in the outgoing packets.

4.2 Experiments and Results

We transmit labelled packets of concurrent multicast sessions at the maximum link speed in our testbed (10 Gbps) from the traffic-generating server to the Yeti Router. We stress Yeti by transmitting traffic that requires copying and rearranging labels for each packet. In every experiment, we attach labels with different sizes to each packet. These labels contain MCT and CPY labels. The MCT label instructs the Yeti Router to duplicate packets on two ports B and C in Figure 4. We report the results for a sample label size of 28 bytes. This is because, as we show in §5, most of the packets have this label size for

traffic engineering and service chaining scenarios in different ISP networks. The CPY labels instruct Yeti to copy 12 and 16 bytes to ports B and C, respectively. We measure the outgoing traffic on port B. The main parameter that we control is the packet size, which we vary from 64 to 1024 bytes. We report three important metrics for the design of high-end routers: packet latency, resource usage, and throughput.

Latency. We report the packet processing latency at port B in Figure 4 when the Yeti Router processes CPY labels (i.e., the worst-case scenario in terms of processing). We measure the latency by timestamping each packet at the traffic generator, and taking the difference between that timestamp and the time the packet is received at port B. We use the Berkeley Extensible Software Switch [15] to timestamp packets. Since it may add overheads while timestamping and transmitting packets, we compare the latency of Yeti processing against the basic forwarding in the same testbed, which is done by matching the fixed-length MAC address. Table 2 shows multiple statistics of the packet latency for both Yeti and unicast forwarding when the packet size is 1,024 bytes. The table shows that the latency of Yeti processing under stress is close to the simple unicast forwarding. For example, the difference of the 95th percentiles of packet latency is only 0.1 μsec when the packet size is 1,024 bytes.

Resource Usage. We measure the resource usage of the packet processing algorithm, in terms of the number of used look-up tables (LUTs) and registers in the Yeti Router, which are generated by the Xilinx Vivado tool after synthesizing and implementing the project. Our implementation uses 12,677 slice LUTs and 1,701 slice registers per port. Relative to the available resources, the used resources are only 3% and 0.2% of the available LUTs and registers, respectively. Thus, Yeti requires small amount of resources while it can forward traffic of many concurrent multicast sessions.

Throughput. In Figure 5, we compare the rate of incoming packets to the Yeti Router versus the rate of packets observed at port B. The figure shows that the numbers of transmitted and received packets per second are the same (i.e., no packet losses). The figure also shows that our algorithm can sustain the required 10 Gbps throughput for all packet sizes.

We realize that core routers have large port density and high speeds. We believe that Yeti can achieve line-rate performance in these routers, because Yeti processes each incoming packet independently and adds small processing latency per packet as shown in Table 2.
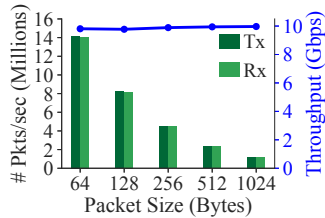
Figure 5: Throughput of received traffic from our testbed.

5 Evaluation using Simulation

We analyze the performance of Yeti and compare it against the closest multicast approaches using simulation.

5.1 Simulation Setup

Simulator. We implemented a Python-based simulator to compare the performance of different multicast systems in large setups using realistic ISP topologies. The simulator has two components. The first acts as the Yeti controller in Figure 1. When this component receives an egress router event, it updates the corresponding multicast graph, and then generates labels using Algorithm 1. The second component simulates the packet processing algorithm in §3.5. The simulator also implements prior systems for comparisons.

ISP Topologies. We use 14 topologies in the Internet Topology Zoo dataset [35]. This dataset represents a wide range of actual ISPs, where the number of routers ranges from 36 to 197, and the number of links ranges from 152 to 486.

Multicast Sessions. We simulate dynamic and diverse multicast sessions. The source of each session is randomly selected from one of the ISP routers. The session bandwidth is randomly chosen from the set {0.5, 1, 2, 5, 10} Mbps, which represents the bandwidth values of different types of applications. The session duration is randomly assigned to a value from {10, 20, 40, 60, 80, 100, 120} minutes. These values reflect a wide range of short to long multicast sessions. In addition, while the session is active, we make its receivers join and leave according to random events generated from a Poisson distribution, where 60% of these are join events and 40% are leave events. We make the receiver join rate 50% higher than the leave rate to incrementally stress the system with more multicast receivers as the time passes.

Each multicast session requires a set of network services, or service chain. We vary the length of the service chain from 3 to 5 as these lengths represent common service usage patterns [1, 42]. To represent practical deployment of services in ISPs, we divide services to essential and supplementary according to their popularity [42]. Essential services such as firewalls are deployed at all ISP locations, whereas supplementary services such as video encoders are only deployed at some of the ISP locations. To stress our system, we set the percentage of ISP locations that provide supplementary

services to only 25%. Each multicast session includes two randomly chosen essential services and the rest of the services are supplementary ones, also randomly chosen.

Since Yeti does not dictate how multicast graphs are computed, we use the algorithms in [12] and [1] to calculate the graphs based on the traffic engineering and service chaining requirements, respectively.

Experiments and Statistics. We simulate the operation of an ISP managing concurrent and dynamic multicast sessions over an extended period of time (24 hours), where about 200,000 sessions are created over the simulation period. Specifically, we first choose one of the 14 ISP topologies and generate the multicast sessions using the characteristics described above. Then, we repeat the experiment for the same ISP topology five times, starting from different seeds for the random distributions. Thus, for each ISP topology, we collect and analyze statistics from about 1M randomly generated, diverse, and dynamic multicast sessions. Then, the whole process is repeated for each of the 14 ISP topologies.

We report the 95-percentile of various performance metrics in the following subsections, as it reflects the performance over extended number of sessions. We present representative samples of our figures, using the ISP topologies with the largest and median numbers of routers. We also present averages and normalized averages (per router) across all ISP topologies, to infer the performance in general settings. When we present the (normalized) averages, we report the standard deviation in each case preceded by $\pm$.

Yeti is the first multicast forwarding system to support service chaining and traffic engineering. Thus, we first compare a simpler version of Yeti against the state-of-art systems for multicast sessions with traffic engineering requirements as these cannot support service chaining. Then, we analyze the performance of Yeti for multicast sessions with traffic engineering and service chaining requirements.

5.2 Yeti vs Stateful and Hybrid Approaches

We compare Yeti versus the closest stateful and hybrid multicast forwarding systems, which are OpenFlow [17] and LIPSIN [25]. We implemented a rule-based multicast forwarding system using OpenFlow [17] (referred to as RB-OF), because rule-based is a general packet processing model that is supported in many networks. The rule-based system is stateful as it installs match-action rules in routers.

LIPSIN is a hybrid approach that encodes the tree link IDs of a session using a Bloom filter. For every link, the LIPSIN controller maintains D link ID tables with different hash values. LIPSIN creates the final label by selecting the table that results in the minimum false positive rate. Since LIPSIN may result in false positives, each router maintains state about incoming links and the label that may result in loops for every session passing through this router. We set the filter size of LIPSIN to the 99th-percentile of the label
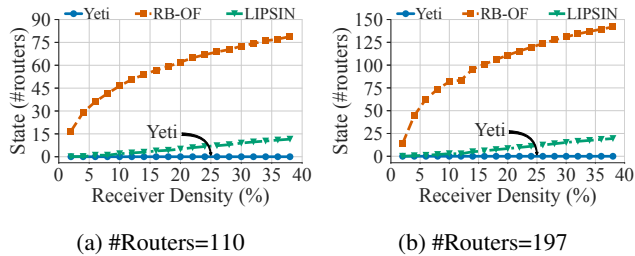
(a) #Routers=110

(b) #Routers=197

Figure 6: State size.

ISP Size	RB-OF	LIPSIN	Yeti	Saving (%)
49	18.9	4.9	1.0	94.7 / 79.7
84	55.9	12.3	1.0	98.2 / 92.0
125	76.8	19.6	1.0	98.7 / 95.0
158	91.2	11.8	1.0	98.9 / 91.5
197	103.3	18.7	1.0	99.0 / 94.7
Norm. Avg. (%)	48±10	9±3	1±0.6	**97.5 / 87.1**

Table 3: Number of routers that need to be updated for each change in the multicast distribution graph. The shown savings in the right most column are relative to RB-OF and LIPSIN, respectively. The averages in the bottom row are computed across all 14 ISP topologies and normalized by the number of routers in each topology, and they represent the average percentage of routers to be updated for each change.

size of Yeti. This enables LIPSIN to encode a large number of links per tree in its labels. We use the same parameters proposed for LIPSIN: we set D to 8 tables and use five hash functions per link. We use Bloom filters and Murmurhash3 hashing functions.

State Size. Figure 6 shows the 95-percentile of the state size per multicast session as the density of receivers varies, which corresponds to the number of core routers needed to maintain state. The results are shown for the median and largest topologies with sizes 110 and 197 routers, respectively. The results for other topologies are similar (shown in Appendix §E).

First, notice that Yeti does not require any state at any core router. In contrast, RB-OF needs to maintain state at each router and that state increases with the topology size as well as the density of receivers in each multicast session. For example, the state size increases from 80 to 130 rules when the receiver density increases from 10% to 30%. LIPSIN, on the other hand, needs to maintain state at up to 20 routers when the receiver density is 40%. In this case, multicast graphs span 50% of the routers. That is, LIPSIN needs to maintain state at up to 20% of the routers in the multicast graph.

State Update Rate. Maintaining state at routers does not only consume their limited SRAM, but also increases the overheads of updating this state at routers when the distribution graph changes [6, 20].

We assess the average number and percentage of routers to be updated when a single multicast tree changes, and show the results for sample ISP topologies with various sizes in Table 3. Recall that the Yeti controller needs to update *one and only one* (ingress) router when a session changes, which is independent of the topology size. The state for RB-OF and LIPSIN, on the other hand, grows with the topology size and number of receivers in the multicast sessions. Thus, as Table 3 shows, RB-OF and LIPSIN controllers need to update up to 103.3 and 19.6 core routers per each distribution graph change, respectively. That is, Yeti reduces the number of routers to be updated by up to 103X and 20X compared to RB-OF and LIPSIN, respectively.

To demonstrate the generality of Yeti performance, we calculate the average percentage of routers in the ISP topology to be updated for each change in the multicast distribution graph, which is taken over all 14 ISP topologies. We present

the results in the last row in Table 3, which show that Yeti reduces the average state update rate by 97.5% and 87.1% compared to RB-OF and LIPSIN, respectively.

In summary, compared to stateful and hybrid approaches, Yeti scales well and can handle dynamic multicast sessions, as it does not require maintaining state at any router, and it significantly reduces the need for frequent state updates.

5.3 Yeti vs A Stateless Approach

We compare Yeti against BIER-TE [3], which is a recent label-based multicast forwarding system. We implemented the basic features of BIER-TE as described in [3], which are the bit positions for the forward-connected, forward-routed and local-decap actions. This means that we *conservatively* report the minimum size of BIER-TE labels for every ISP topology. Since Yeti and BIER-TE are stateless and both use labels, we only analyze the label size and its imposed total overhead.

We first asses the label size per packet for multiple receiver densities. We present the CDF of the label size for the median and largest topologies in Figure 7; the results for other topologies are given in Appendix §E. Figure 7 indicates that the label size for Yeti is much smaller than that of BIER-TE in practical scenarios. For example, for the topology with the median number of routers (110 routers), Figure 7a, and receiver density of 30%, Yeti reduces the label size for 50% of the packets by 91.6% compared to BIER-TE. Moreover, the label size in Yeti for 90% of the packets is less than 19 bytes, while the label size of BIER-TE is 64 bytes. For the largest topology in our dataset (197 routers) and for receiver density of 30%, Figure 7b shows that the label size in Yeti is 15X smaller than BIER-TE for 50% of the packets.

We further analyze the behavior of Yeti and the dynamics of its label size as packets traverse the network. In Figure 8,
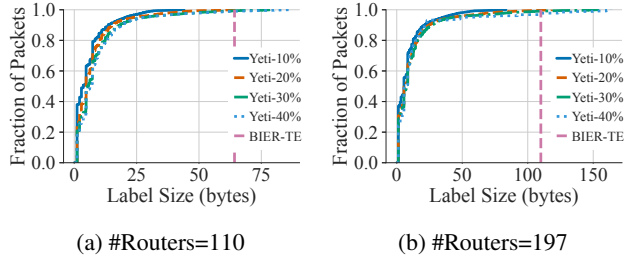
(a) #Routers=110   (b) #Routers=197

Figure 7: Label size CDF for different receiver densities.

(a) #Routers=110   (b) #Routers=197

Figure 8: Label size as packets traverse the network.

we plot the average label size for Yeti and BIER-TE versus the number of hops from source, for the median and largest topologies, more results for other topologies are given in Appendix §E. The figure shows that the label size for BIER-TE depends on the topology size, it is 64 and 110 bytes, for the median and largest topologies, respectively. In contrast, the label size in Yeti decreases quickly as the packet moves away from the source. For example, in Figure 8a, the label size of Yeti is reduced by 16.9% and 67.8% after traversing 1 and 5 hops, respectively. The label size of Yeti becomes smaller than that of BIER-TE after traversing 2 hops only, and Yeti reduces the label size by 87.5% after traversing the first 50% of the hops.

Finally, we assess the total end-to-end label overhead of Yeti and BIER-TE, which we define as the label size multiplied by the number of network hops the packet traverses; this is the area under the curves in Figure 8. Table 4 summarizes the label overhead in bytes for multiple sample topologies. The results show that Yeti achieves substantial savings, up to 70.2%, compared to BIER-TE. In addition, we report the average label overhead per router across all 14 ISP topologies. On average, Yeti needs only 4 bytes per router to forward packets of multicast sessions, whereas BIER-TE requires 11.4 bytes per router, that is Yeti achieves an average saving of 65.3% in the label overhead.

In summary, the label size in Yeti quickly decreases as packets move towards the multicast destinations, because routers copy only a subset of labels for every branch. This results in substantial savings in the label overheads compared to the closest label-based multicast forwarding system, BIER-TE. In addition, BIER-TE cannot satisfy the service chaining requirements for multicast traffic, while Yeti can.

5.4 Analysis of Yeti

We analyze the performance of Yeti in terms of effectiveness of FSP labels, label size, processing overheads and running time to satisfy various forwarding requirements.

Effectiveness of FSP Labels. We study the importance of the proposed FSP label type. Recall that a single FSP label represents multiple routers in a path segment if that segment is on the shortest path. To show the importance of FSP label
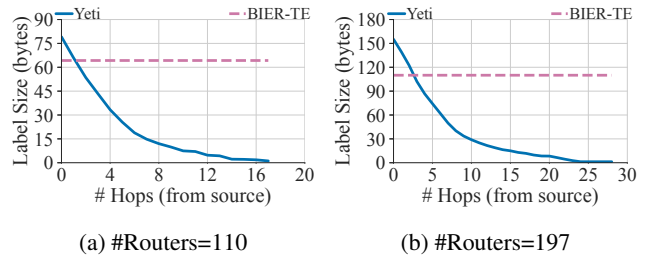
ISP Size	BIER-TE	Yeti	Saving (%)
49	299.3	89.3	70.2
84	1,283.3	508.5	60.4
125	1,761.5	588.5	66.6
158	3,123.0	1,176.2	62.3
197	3,190.0	1,062.6	66.7
Norm. Avg. (bytes/router)	11.4±4.8	4.0±1.8	**65.3**

Table 4: Label overhead in bytes for Yeti and BIER-TE.

across different topologies, we plot the average FSP saving for sample topologies in Figure 9a as well as the average over all ISP topologies. We observe consistent savings across the topologies which range from 4 to 9 routers per FSP label. That is, one FSP label saves 4–9 other labels, reducing the label overhead by up to 9X. The average FSP saving is 5.8±1.8 routers.

Next, we asses the FSP savings to satisfy service chaining requirements with different chain lengths in Table 5. As the results show, FSP labels efficiently encode path segments in the distribution graphs. For instance, an FSP label can encode about 6 routers on average for typical chain lengths.

Label Size. In Figure 9b, we plot the label size of Yeti versus the number of hops from source for service chaining requirements. The figure shows the results for the median topology of size 110. The results indicate that the label size of Yeti quickly decreases for all considered chain lengths as packets traverse towards the destinations. For instance, the label size decreases by 23% after traversing the first 10 hops when the chain length is 4. In addition, although the used routing policy [1] increases the number of hops to satisfy all service chaining requirements, Yeti is able to encode these large graphs in relatively small labels.

Processing Overhead. In Figure 10a, we plot the number of copy operations per packet versus the receiver density for multiple topology sizes to realize traffic engineering. The results for other ISP topologies are plotted in Appendix §E. The figure shows that the additional work per packet is small. The number of copy operations increases as the receiver density increases because the multicast distribution graphs have
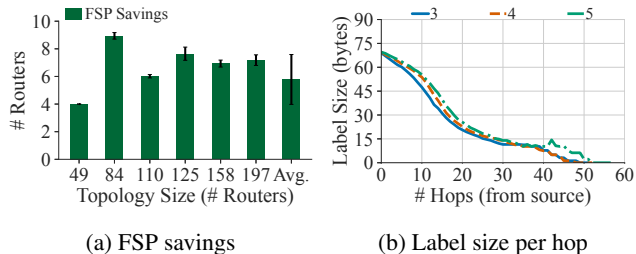
(a) FSP savings

(b) Label size per hop

Figure 9: Analysis of FSP savings and label size of Yeti to satisfy traffic engineering and service chaining requirements.

(a) # copy operations

(b) Distribution of operations

Figure 10: Analysis of processing overheads of Yeti.

ISP	Service Chain Length		
Size	3	4	5
49	4.2±0.4	4.0±0.0	4.0±0.0
84	11.2±1.7	11.0±1.4	11.0±1.4
125	9.2±1.2	9.0±1.3	9.2±1.2
158	8.0±1.1	8.0±1.1	8.0±1.1
197	7.2±1.2	7.2±1.2	7.2±1.2
Avg.	6.4±2.2	6.3±2.2	6.4±2.2

Table 5: Number of traversed routers per FSP label to satisfy service chaining.

more branches in this case. However, the processing overhead increases slowly. For instance, in the 84-router topology, the average number of copy operations increases from 0.4 to 0.62 per packet when the receiver density increases from 5% to 38%. This pattern applies for other topologies as well. That is, Yeti routers scale in terms of processing as the network load increases.

We next present the distribution of FSP, FTE, MCT and CPY labels per router for the five topologies in Figure 10b. The figure shows that the fraction of processed CPY labels (the most expensive) per Yeti router across all sessions is small compared to other label types. For example, only 17% of the labels being processed at a Yeti router are CPY labels for the largest topology of 197 routers.

For service chaining, Yeti incurs a similar distribution of operations to Figure 10b. For instance, the fraction of processed CPY labels is 18.7% for the topology of size 110 when the chain length is 3. For a longer chain of length 5, the fraction of CPY labels increases slightly to 21% to satisfy all service chains. On average, the fractions of CPY labels per ISP topology are 19.2%±2.8%, 19.6%±2.8%, and 19.6%±2.3% for chain lengths of 3, 4 and 5, respectively. We present the average number of copy operations and distribution of all operations in Appendix §E, where the averages are taken over chain lengths.

Running Time of Yeti Controller. We ran the proposed CRE-ATELABELS algorithm on a workstation with four 3.3 GHz cores and 32 GB of memory, and we measured the running time of creating labels per graph update at the controller. The running time varied from 4 msec to 10 msec based on the topology size. For the largest topology of size 197, the controller spends only about 10 msec per graph update to create the labels for the largest session. Thus, the proposed label creation algorithm is practical, can run on commodity servers, and it supports frequent graph updates and network dynamics.

In summary, Yeti imposes small label and processing overheads while satisfying service chaining and traffic engineering requirements.

6 Conclusions

We proposed an efficient, stateless, multicast forwarding system called Yeti that implements generalized multicast graphs in ISP networks. Unlike current rule-based multicast systems, Yeti does not require maintaining any state at routers. And unlike other label-based multicast systems, Yeti can direct traffic on arbitrary network paths to meet traffic engineering and service chaining requirements while reducing the label size significantly. The novel aspects of Yeti include (1) supporting general traffic forwarding requirements, (2) guaranteeing correctness, (3) composing small labels, and (4) processing labels efficiently at routers. We implemented Yeti in a programmable router and evaluated its performance. Our experiments show that Yeti can achieve line-rate performance while using a small amount of hardware resources. In addition, we conducted extensive simulations using real ISP topologies, and compared it versus the state-of-art approaches. Our results show that Yeti outperforms the other approaches by wide margins for all considered metrics.

Acknowledgments

References

[1] K. Diab, C. Lee, and M. Hefeeda. Oktopus: Service chaining for multicast traffic. In *Proc. of IEEE ICNP'20*, pages 1–11, Madrid, Spain, October 2020.

[2] Anja Feldmann, Oliver Gasser, Franziska Lichtblau, Enric Pujol, Ingmar Poese, Christoph Dietzel, Daniel Wagner, Matthias Wichtlhuber, Juan Tapiador, Narseo Vallina-Rodriguez, Oliver Hohlfeld, and Georgios Smaragdakis. The lockdown effect: Implications of the covid-19 pandemic on internet traffic. In *Proc. of ACM IMC'20*, page 1–18, Virtual Event, October 2020.

[3] Toerless Eckert, Gregory Cauchie, and Michael Menth. Tree Engineering for Bit Index Explicit Replication (BIER-TE). Internet-Draft draft-ietf-bier-te-arch-08, Internet Engineering Task Force, July 2020. Work in Progress.

[4] Clarence Filsfils, Stefano Previdi, Les Ginsberg, Bruno Decraene, Stephane Litkowski, and Rob Shakir. Segment Routing Architecture. RFC 8402.

[5] B. Ren, D. Guo, G. Tang, X. Lin, and Y. Qin. Optimal service function tree embedding for nfv enabled multicast. In *Proc. of IEEE ICDCS'18*, pages 132–142, Vienna, Austria, July 2018.

[6] Jiaqi Zheng, Bo Li, Chen Tian, Klaus-Tycho Foerster, Stefan Schmid, Guihai Chen, and Jie Wu. Scheduling congestion-free updates of multiple flows with chronicle in timed sdns. In *Proc. of IEEE ICDCS'18*, pages 12–21, Vienna, Austria, July 2018.

[7] S. H. Chiang, J. J. Kuo, S. H. Shen, D. N. Yang, and W. T. Chen. Online multicast traffic engineering for software-defined networks. In *Proc. of IEEE INFOCOM'18*, pages 414–422, Honolulu, HI, April 2018.

[8] Aravindh Raman, Gareth Tyson, and Nishanth Sastry. Facebook (A)Live?: Are Live Social Broadcasts Really Broadcasts? In *Proc. of WWW'18*, page 1491–1500, Lyon, France, April 2018.

[9] Victor Heorhiadi, Sanjay Chandrasekaran, Michael K. Reiter, and Vyas Sekar. Intent-driven composition of resource-management sdn applications. In *Proc. of ACM CoNEXT'18*, pages 86–97, Heraklion, Greece, 2018.

[10] IJsbrand Wijnands, Eric C. Rosen, Andrew Dolganow, Tony Przygienda, and Sam Aldrin. Multicast Using Bit Index Explicit Replication (BIER). RFC 8279.

[11] L. H. Huang, H. C. Hsu, S. H. Shen, D. N. Yang, and W. T. Chen. Multicast traffic engineering for software-defined networks. In *Proc. of IEEE INFOCOM'16*, pages 1–9, San Francisco, CA, April 2016.

[12] M. Huang, W. Liang, Z. Xu, W. Xu, S. Guo, and Y. Xu. Dynamic routing for network throughput maximization in software-defined networks. In *Proc. of IEEE INFOCOM'16*, pages 1–9, San Francisco, CA, April 2016.

[13] Bill Fenner, Mark J. Handley, Hugh Holbrook, Isidor Kouvelas, Rishabh Parekh, Zhaohui (Jeffrey) Zhang, and Lianshu Zheng. Protocol Independent Multicast - Sparse Mode: Protocol Specification. RFC 7761.

[14] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. Moongen: A scriptable high-speed packet generator. In *Proc. of ACM IMC'15*, pages 275–287, Tokyo, Japan, October 2015.

[15] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. Softnic: A software nic to augment hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015.

[16] S. H. Shen, L. H. Huang, D. N. Yang, and W. T. Chen. Reliable multicast routing for software-defined networks. In *Proc. of IEEE INFOCOM'15*, pages 181–189, Hong Kong, China, April 2015.

[17] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. *Proc. of the IEEE*, 103(1):14–76, Jan 2015.

[18] Renaud Hartert, Stefano Vissicchio, Pierre Schaus, Olivier Bonaventure, Clarence Filsfils, Thomas Telkamp, and Pierre Francois. A declarative and expressive approach to control forwarding paths in carrier-grade networks. In *Proc. of ACM SIGCOMM'15*, pages 15–28, London, United Kingdom, 2015.

[19] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore. Netfpga sume: Toward 100 gbps as research commodity. *IEEE Micro*, 34(5):32–41, September 2014.

[20] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. Dynamic scheduling of network updates. In *Proc. of ACM SIGCOMM'14*, pages 539–550, Chicago, IL, August 2014.

[21] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, July 2014.

[22] X. Chen, M. Chen, B. Li, Y. Zhao, Y. Wu, and J. Li. Celerity: A low-delay multi-party conferencing solution. *IEEE Journal on Selected Areas in Communications*, 31(9):155–164, September 2013.

[23] Bob Thomas, IJsbrand Wijnands, Ina Minei, and Kireeti Kompella. LDP Extensions for Point-to-Multipoint and Multipoint-to-Multipoint Label Switched Paths. RFC 6388.

[24] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: a distributed messaging system for log processing. In *Proc. of ACM Workshop on Networking Meets Databases (NetDB'11)*, Athens, Greece, June 2011.

[25] Petri Jokela, András Zahemszky, Christian Esteve Rothenberg, Somaya Arianfar, and Pekka Nikander. Lipsin: Line speed publish/subscribe inter-networking. In *Proc. of ACM SIGCOMM'09*, pages 195–206, Barcelona, Spain, August 2009.

[26] T. W. Cho, M. Rabinovich, K. K. Ramakrishnan, D. Srivastava, and Y. Zhang. Enabling content dissemination using efficient and scalable multicast. In *Proc. of IEEE INFOCOM'09*, pages 1980–1988, Rio de Janeiro, Brazil, April 2009.

[27] V. Gopalakrishnan, B. Bhattacharjee, K. K. Ramakrishnan, R. Jana, and D. Srivastava. Cpm: Adaptive video-on-demand with cooperative peer assists and multicast. In *Proc. of IEEE INFOCOM'09*, pages 91–99, Rio de Janeiro, Brazil, April 2009.

[28] Bradley Cain, Dr. Steve E. Deering, Bill Fenner, Isidor Kouvelas, and Ajit Thyagarajan. Internet Group Management Protocol, Version 3. RFC 3376.

[29] C. Diot, B. N. Levine, B. Lyles, H. Kassem, and D. Balensiefen. Deployment issues for the ip multicast service and architecture. *IEEE Network*, 14(1):78–88, January 2000.

[30] Wen-Tsuen Chen, Pi-Rong Sheu, and Yaw-Ren Chang. Efficient multicast source routing scheme. *Computer Communications*, 16(10):662–666, 1993.

[31] Stephen E. Deering and David R. Cheriton. Multicast routing in datagram internetworks and extended lans. *ACM Transactions on Computer Systems*, 8(2):85–110, May 1990.

[32] AWS Announces Nine New Compute and Networking Innovations for Amazon EC2. https://bloom.bg/2t1N9py. [Online; accessed February 2022].

[33] Run IP Multicast Workloads in the Cloud Using AWS Transit Gateway. https://go.aws/2RT6stz. [Online; accessed February 2022].

[34] NetFPGA SUME Reference Learning Switch Lite. https://bit.ly/2UrUFlx. [Online; accessed February 2022].

[35] The Internet Topology Zoo. http://www.topology-zoo.org/dataset.html. [Online; accessed February 2022].

[36] Apache ActiveMQ. http://activemq.apache.org. [Online; accessed February 2022].

[37] Bt iptv (youview). https://bit.ly/3ssCvTz. [Online; accessed February 2022].

[38] Multicast Command Reference for Cisco ASR 9000 Series Routers. https://bit.ly/3AaVGDQ. [Online; accessed February 2022].

[39] RabbitMQ. http://www.rabbitmq.com. [Online; accessed February 2022].

[40] U-verse tv. https://bit.ly/3HLhfyL. [Online; accessed February 2022].

[41] Zuckerberg really wants you to stream live video on Facebook. https://bit.ly/2v6uHqF. [Online; accessed February 2022].

[42] Benoit Donnet, Korian Edeline, Iain R. Learmonth, and Andra Lutu. Middlebox classification and initial model. https://bit.ly/3dZelXV. [Online; accessed February 2022].

Appendix A   Correctness of Yeti

Theorem 1 (Correctness). *Yeti forwards packets on and only on links that belong to the multicast graph.*

Proof. Yeti guarantees correctness by creating an ordered set of Yeti labels for the given graph at the controller using the ENCODEGRAPH algorithm. Recall that the algorithm first creates a tree to represent the services needed before reaching the destinations. A node in that calculated tree consists of router ID v and sub-sequence of services $\mathbb{S}$. Every node appears only once in the tree (by construction). This means that the tree has no cycles.

The label creation algorithm traverses the tree to calculate the final labels. Within every path with provided services $\mathbb{S}$, the order and type of created labels represent how packets should be forwarded in the data plane. This is detailed as follows. FSP labels do not result in incorrect forwarding because: (1) an FSP label with ID v is only added when a tree node v is traversed by the CREATELABELS algorithm, (2) since every node with router ID v and services $\mathbb{S}$ is traversed once and only once by the algorithm, only a single FSP v can be added to the labels representing that node with services $\mathbb{S}$, (3) in the data plane, the router with ID v removes the FSP v label. Thus, no subsequent routers in along the path with same services can process that label and transmit the packet back to v, and (4) since the traversal starts from the source, if

a node *u* precedes node *v* in the tree, the algorithm guarantees that *u* is traversed before *v*. Thus, there is no label to forward packets back to *u*.Similar properties are guaranteed for FTE labels in terms of links. Moreover, attaching multiple FTE labels does not result in incorrect forwarding. Otherwise, the given tree has loops, or routers do not remove FTE labels.

For MCT and CPY labels, since the CREATELABELS algorithm recursively creates the labels for each branch, the same guarantees apply within a single branch for branching points. In addition, routers in one branch do not forward packets to routers in other branches. This is because (1) the given tree is a proper one (i.e., has no cycles), and (2) every router in a given branch processes the subset of labels duplicated for that branch using the CPY labels. □

Appendix B Practical Considerations of Yeti

Multicast across ISPs. The description of Yeti thus far has focused on offering a scalable multicast service within a *single* ISP. Extending Yeti to multiple ISPs can be done in multiple ways. For example, a content provider could have separate agreements with different ISPs to serve clients within these ISPs, where each ISP runs its multicast service independently from the others. In this case, a separate feed of the multicast session traffic is provided from the content provider to each ISP. Agreements between major content providers, e.g., Facebook and Netflix, and large ISPs are not uncommon. Another way of extending Yeti to multiple ISPs is through tunneling, where a tunnel is established between an egress router of an ISP to an ingress router of another ISP. The ingress router of the second ISP would attach labels created by the controller of that ISP. While the tunneling approach does not reveal the internal network details of ISPs to each other, which is important in practice, it does require collaboration among ISPs to establish tunnels among some routers.

Incremental Deployment. An ISP may have some legacy routers that are not programmable and thus cannot run the packet processing algorithm of Yeti. There are multiple options that Yeti can still function in this situation, albeit with some workaround and minor overheads. First, Yeti is general and can support arbitrary multicast graphs. Thus, a possible solution is to modify the multicast graphs to avoid going through legacy routers. The multicast graphs is an *input* to our label creation algorithm, and thus the computed labels will not direct traffic through legacy routers. If a legacy router cannot be avoided, a tunnel can be created between the router immediately before the legacy router and each router following it has multicast destinations.

Appendix C Illustrative Example

We present a simple example to illustrate all steps of the proposed approach. Figure 11 shows the multicast tree of the session in Figure 1, where solid arrows indicate the graph links. The dotted line is the shortest path that the ISP avoids because it is over-utilized. Router IDs and used interface IDs are shown in the figure. The number of core routers and maximum interface count are 12 and 5, respectively. Thus, the label sizes (in bits) are 8, 8, 7 and 5 for MCT, CPY, FSP and FTE, respectively (Table 1).

The controller generates the shown labels using the ENCODEGRAPH algorithm as follows. First, the algorithm creates three FSP labels to encode path segments to routers 2, 7 and 4. Notice that the most significant bit in each of these labels is set to one as these nodes provide services a, b and c.

The algorithm then generates MCT 1-00110 to duplicate packets on interfaces 2 and 3 at router 4. Since the children 3 and 7 have core children, the algorithm sets the most significant bit in the MCT label to one, and creates two CPY labels for branches A and B. In branch B, the recursive call of Algorithm 1 creates labels for the path segment {3, 6, 5, 8} as follows. First, the algorithm appends routers 3, 6 and 5 to *pth_seg* because each has one core child (Line 13, Algorithm 1). When the algorithm reaches 8 (which has two core children), the algorithm appends it to *pth_seg* (Line 25, Algorithm 1) and creates labels for the path segment and the branching point at router 8. For the path segment, since link (3, 6) is not on the shortest path from 3 to 8, the algorithm creates FTE 011 to forward packet on interface 3 at router 3. In addition, the algorithm creates FSP 0-1000 to forward packets from 6 to 8, because the links (6, 5) and (5, 8) are on the shortest path between 3 and 8.

We describe the packet processing algorithm at representative routers. The dark labels in Figure 11 are the ones that are processed at given routers. When router 2 receives FSP 1-0010, it decides that the packet needs to be processed by service a. So, it removes the label and forwards the packet to the corresponding datacenter. When the processing is done, router 2 receives a packet which its first label is FSP 1-0111. Thus, it forwards the packet on the shortest path to router 7. Notice that router 4 does not remove FSP 1-0111 as it is not destined for it. Then, routers 7 and 4 receives packets where the contents of the FSP labels are their router IDs. After service c processes the packet, router 4 processes MCT 1-00110 by duplicating the packet on interfaces 2 and 3, and copying specific byte ranges using the CPY labels. Router 3 forwards the packet on interface 3. Router 8 removes FSP and MCT labels and duplicates the packet to routers 11 and 12. Since the packet has no labels at router 11 and 12, they transmit it to egress routers.

Appendix D Implementation of Yeti using P4

P4 is a data plane programming language that is getting popular due to its flexibility. Thus, we show that Yeti can be implemented in P4 switches. We implemented Yeti using the Intel P4 software development environment (SDE) version 9.5.0.
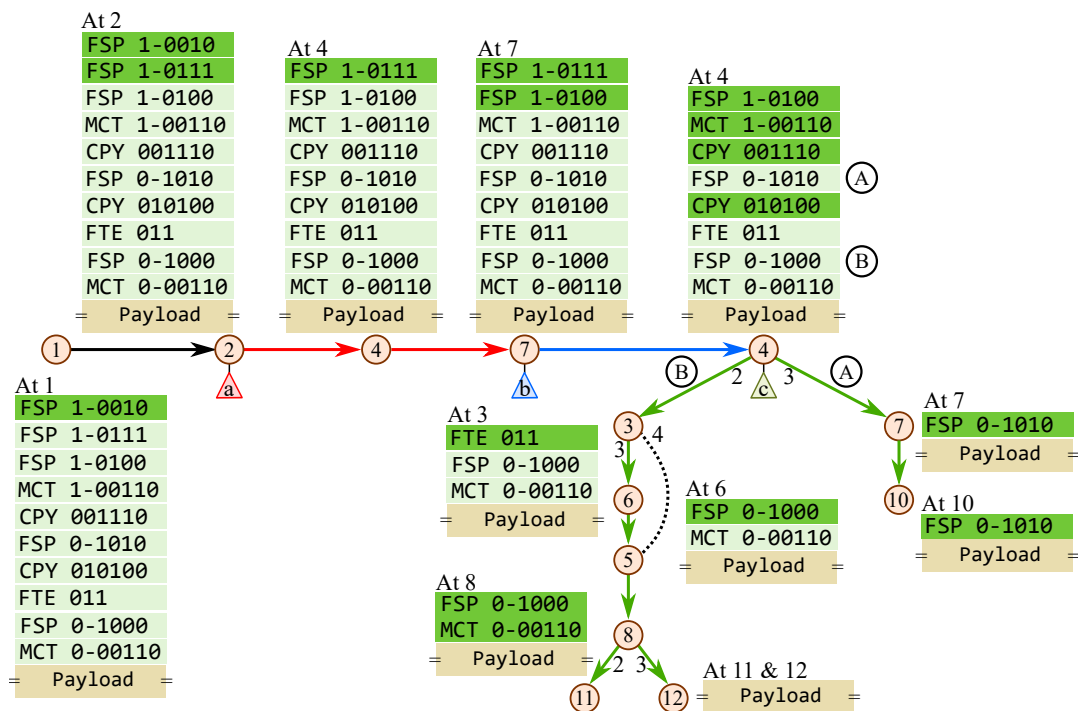

Figure 11: Illustrative example of how labels in Yeti represent the multicast tree in Figure 1.
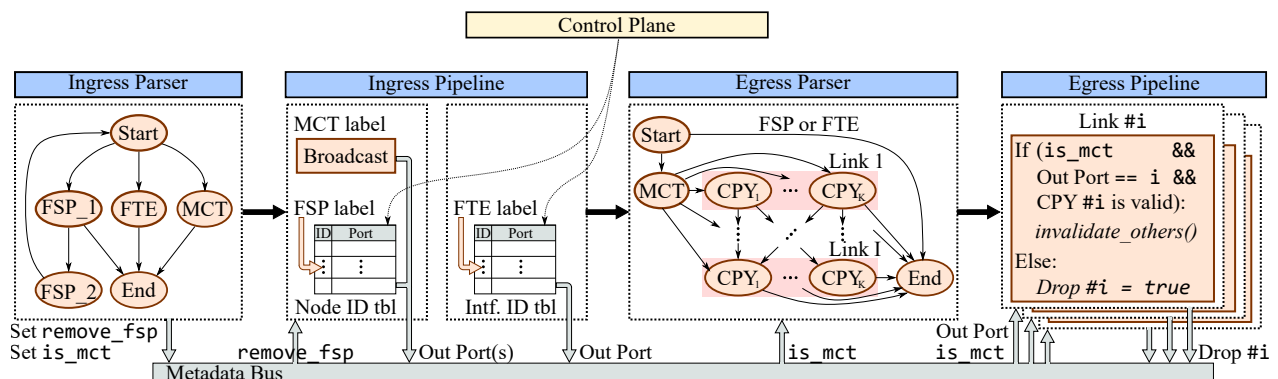

Figure 12: Design of Yeti in P4 switches.

We used the provided tools such as the P4 compiler (called `bf-p4c`) and switch model integrated with the SDE to realize our ideas and validate our implementation. We note that P4 programs implemented in open-source software switches, e.g., `bmv2`, may not necessarily work on actual hardware switches, because of the potential mismatch between the available physical resources in actual switches and the assumed resources in software switches. This is not the case for the Intel SDE, since its P4 compiler produces code for actual Tofino switches (which are also manufactured by Intel).

There are two main challenges in implementing Yeti using P4. First, current Tofino switches do not provide programmable primitives to implement new multicast systems. This is because the packet replication engine is implemented as a fixed-function block. Second, the current `bf-p4c` compiler does not support variable-length fields (i.e., `varbit`) needed to parse and process CPY labels. We made three design choices to address these challenges. We first divided the packet processing between ingress and egress pipelines to realize a programmable multicast primitive. Second, we relied on the available resources and programmable capabilities of the parsers to reduce the processing in ingress/egress stages. Finally, we explicitly unrolled the parsing and processing of a CPY label as an array of items.

Figure 12 illustrates the high-level design of our P4 implementation. Upon a packet arrival, the ingress parser processes FSP, FTE and MCT labels as follows. For an FSP label, the parser reads the included node ID, and parses the labels again if the parsed node ID is the same as the router ID. In this case, the parser sets a metadata field `remove_fsp` to be used by the ingress pipeline. Recall that the Yeti controller never creates two consecutive FSP labels (§3.4 and §3.6). Therefore, the parser always terminates. In the case of FTE and MCT labels, the parser reads their contents. In addition, it sets a metadata field `is_mct` when it parses an MCT label.

The ingress pipeline contains two tables to maintain node and interface IDs, and spans two stages. In the first stage, the algorithm processes an FSP label by reading an entry from the node ID table that matches the label content, and setting the outgoing port accordingly. The algorithm also removes the FSP label if the metadata field `remove_fsp` is set. The algorithm processes MCT labels, in the same stage, by duplicating the packet to all outgoing ports. The FTE label processing is done in the second stage, and it is similar to that of FSP. However, the algorithm always removes FTE labels.

The egress parser and pipeline are used only to handle MCT and CPY labels. For each duplicated packet, the egress parser extracts the contents of MCT label when the metadata field `is_mct` is set. The parser then reads CPY labels sequentially based on the content of MCT label and offsets in CPY labels. Specifically, for each CPY label, the parser extracts its offset and content. The content is read based on the offset value, and represented as an array of up to K multiples of B bits to emulate `varbit<K×B>`. When the parser is done, the egress pipeline identifies which outgoing port should transmit what CPY label based on the egress port number and validity of the CPY label. The MCT and remaining CPY labels are removed.

We did not have a physical Tofino switch in our lab to conduct measurement experiments at the time of conducting this research. We thus validated our implementation by writing and running multiple test cases, and sending and receiving packets to and from the Tofino switch model process. When we run a test case, we insert the required entries into node and link IDs tables. We send labeled packets using `scapy`, and verify the reception of outgoing packets based on the attached Yeti labels. A test case succeeds if all packets were received on and only on expected ports with expected headers.

Appendix E   Additional Simulation Results

This appendix includes more figures and results from our simulation.

Figure 13 shows the state size for all 12 ISP topologies. The figure indicates that Yeti provides a scalable multicast service as it does not require any state at any router.

Figures 14–15 show the label size of Yeti versus BIER-TE. Compared to BIER-TE, the figures show that Yeti reduces the label size significantly across receiver densities and as packets traverse the network.

Figures 16–19 indicate that Yeti impose small label and processing overheads when supporting service chaining and traffic engineering requirements.
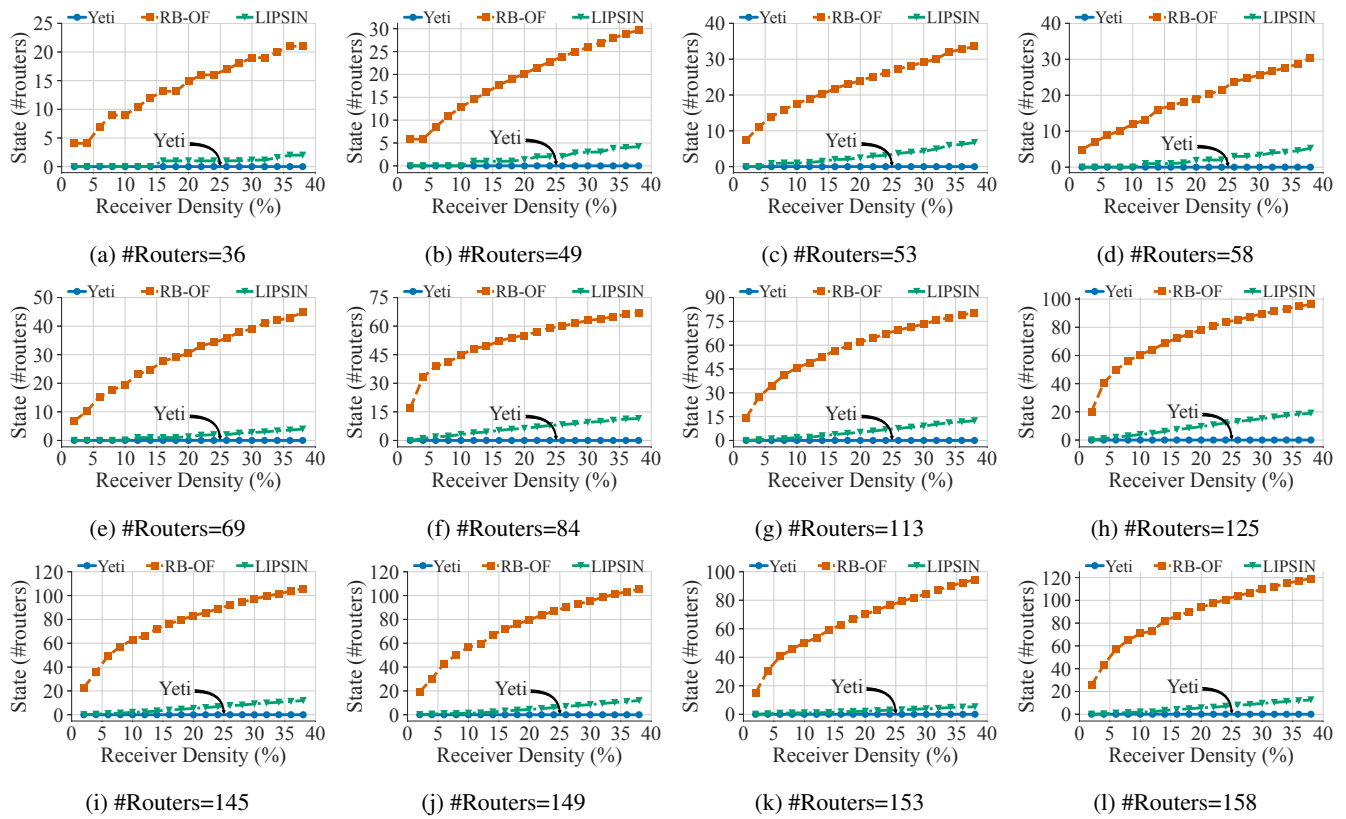
Figure 13: State size for the considered ISP topologies.

(a) #Routers=36   (b) #Routers=49   (c) #Routers=53   (d) #Routers=58

(e) #Routers=69   (f) #Routers=84   (g) #Routers=113   (h) #Routers=125

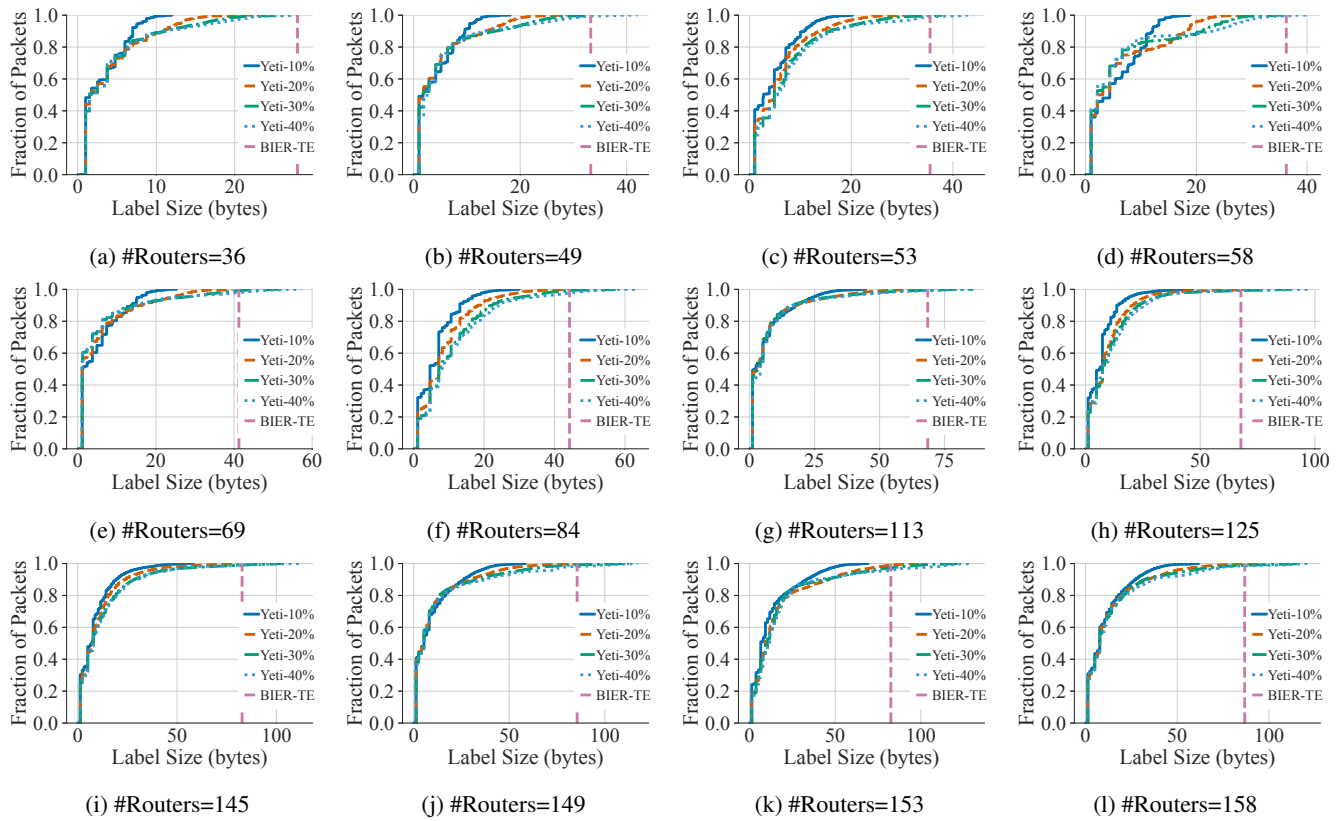(i) #Routers=145   (j) #Routers=149   (k) #Routers=153   (l) #Routers=158

Figure 14: Label size CDF for different ISP topologies.
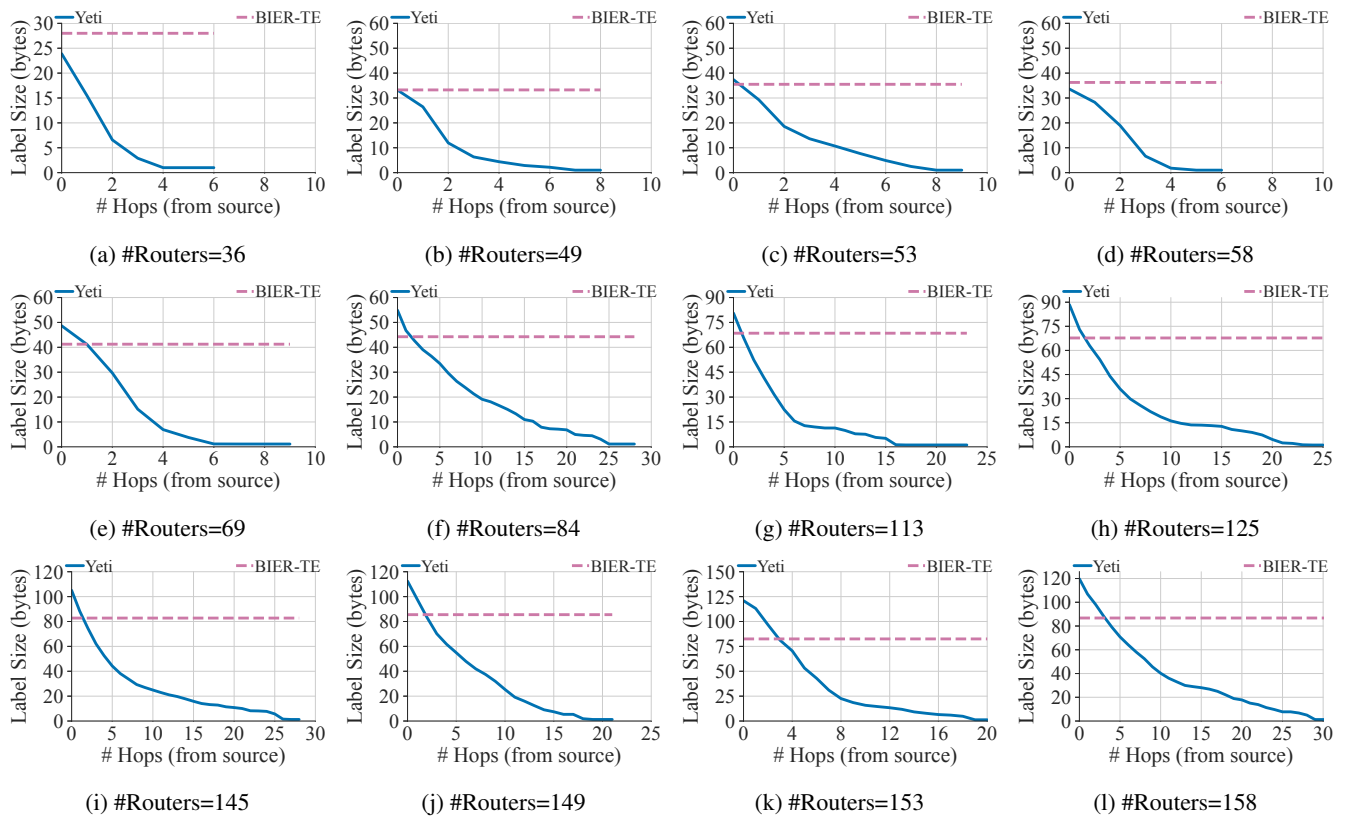
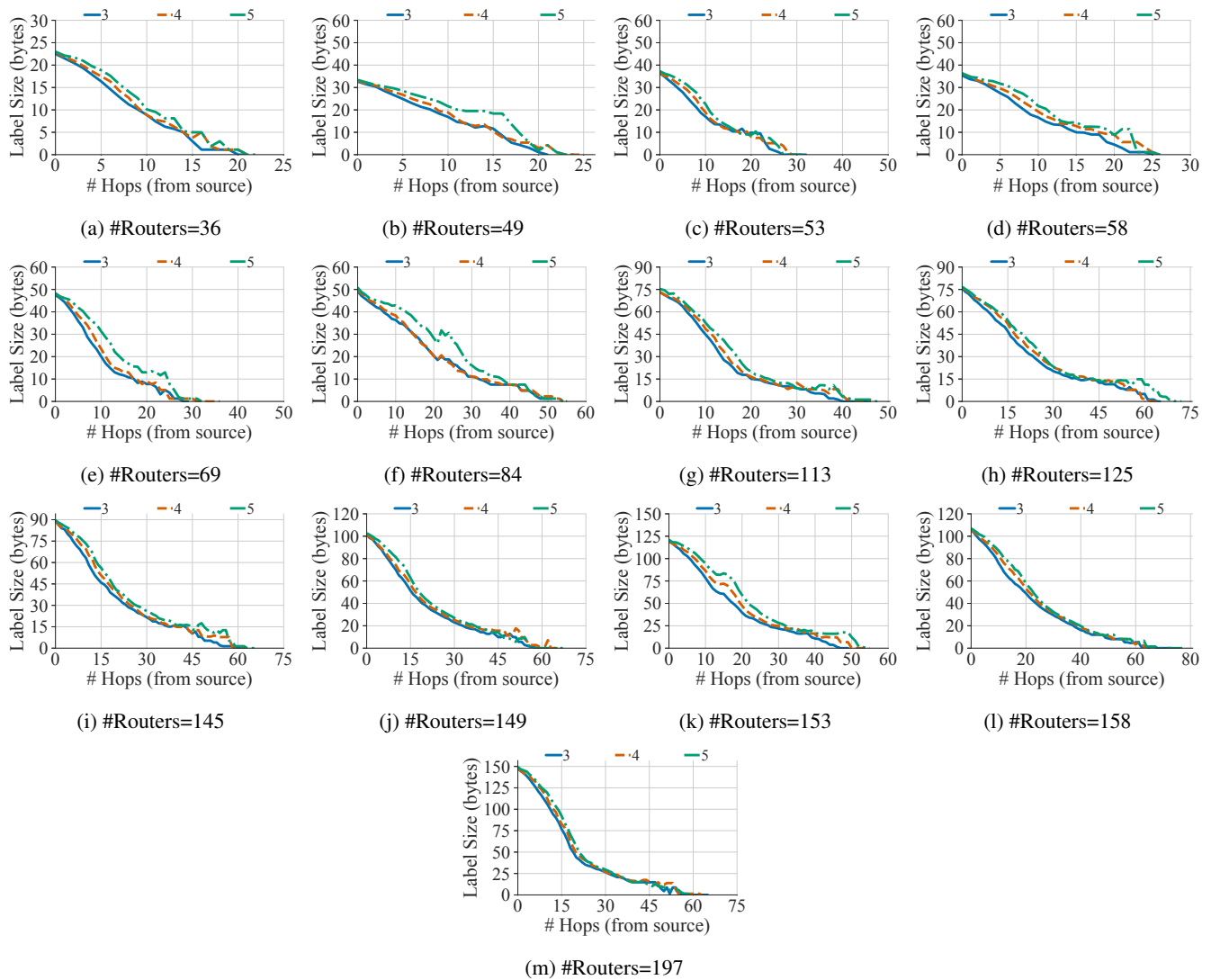Figure 15: Label size of Yeti versus BIER-TE as packets traverse the network.

Figure 16: Analysis of label size of Yeti to satisfy service chaining requirements.
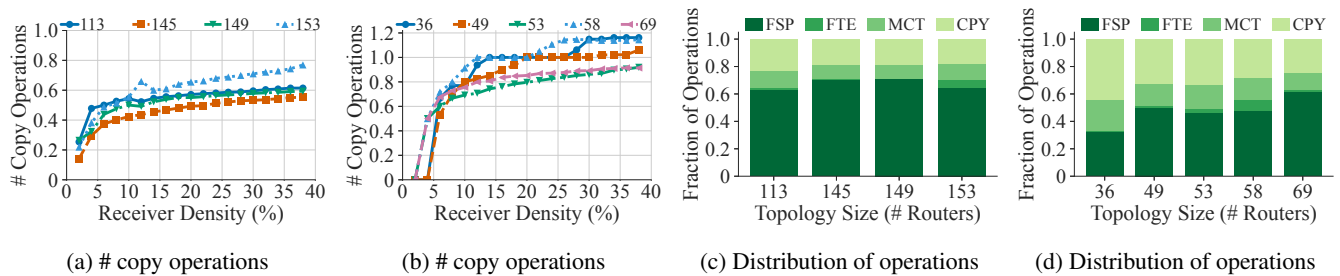
Figure 17: Analysis of processing overheads of Yeti for different ISP topologies.
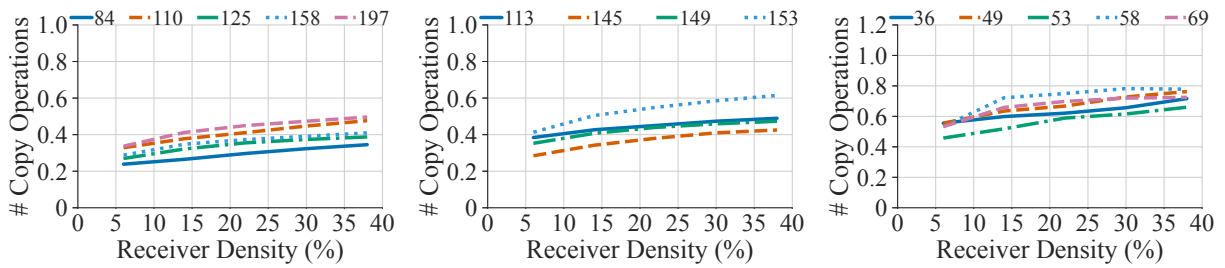
Figure 18: # copy operations for different ISP topologies to support service chaining.
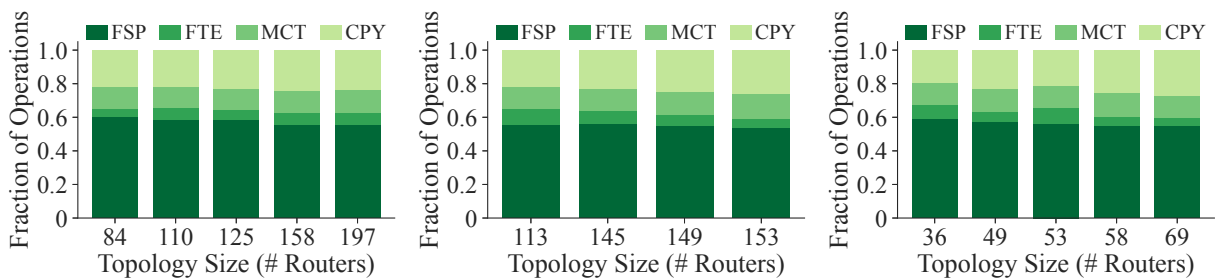
Figure 19: Distribution of FSP, FTE, MCT and CPY operations to support service chaining.

cISP: A Speed-of-Light Internet Service Provider

Debopam Bhattacherjee* [1], Waqar Aqeel* [2], Sangeetha Abdu Jyothi[3,4], Ilker Nadi Bozkurt[2†], William Sentosa[5], Muhammad Tirmazi[6], Anthony Aguirre[7], Balakrishnan Chandrasekaran[8], P. Brighten Godfrey[5,9], Gregory Laughlin[10], Bruce Maggs[2,11], Ankit Singla[1]

[1]ETH Zürich, [2]Duke University, [3]UC Irvine, [4]VMware Research, [5]UIUC, [6]Harvard University, [7]UC Santa Cruz, [8]VU Amsterdam, [9]VMware, [10]Yale University, [11]Emerald Technologies

Abstract

Low latency is a requirement for a variety of interactive network applications. The Internet, however, is not optimized for latency. We thus explore the design of wide-area networks that move data at nearly the speed of light in vacuum. Our **cISP** design augments the Internet's fiber with free-space microwave wireless connectivity over paths very close to great-circle paths. cISP addresses the fundamental challenge of simultaneously providing ultra-low latency while accounting for numerous practical factors ranging from transmission tower availability to packet queuing. We show that instantiations of cISP across the United States and Europe would achieve mean latencies within 5% of that achievable using great-circle paths at the speed of light, over medium and long distances. Further, using experiments conducted on a nearly-speed-of-light algorithmic trading network, together with an analysis of trading data at its end points, we show that microwave networks are *reliably* faster than fiber networks even in inclement weather. Finally, we estimate that the economic value of such networks would substantially exceed their expense.

1 INTRODUCTION

User experience in many interactive network applications depends crucially on achieving low latency. Even seemingly small increases in latency can negatively impact user experience, and, subsequently, revenue for service providers: Google, for example, quantified the impact of an additional 400 ms of latency in search results as 0.7% fewer searches per user [18]. Further, wide-area latency is often the bottleneck, as Facebook's analysis of over a million requests found [21]. Indeed, content delivery networks (CDNs) present latency reduction and its associated increase in conversion rates as one of the key value propositions of their services, citing, e.g., a 1% loss in sales per 100 ms of latency for Amazon [2]. In spite of the significant impact of latency on performance and user experience, the Internet is not designed to treat low latency as a primary objective. This is the problem we address: reducing latencies over the Internet to the lowest possible.

The best achievable latency between two points along the surface of the Earth is determined by their geodesic distance divided by the speed of light, c. Latencies over the Internet, however, are usually much larger than this minimal "c-latency": recent measurement work found that fetching even small amounts of data over the Internet typically takes $37\times$ longer than the c-latency, and often, more than $100\times$ longer [16]. This delay comes from the many round-trips between the communicating endpoints, due to inefficiencies in the transport and application layer protocols, and from each round-trip itself taking 3-4$\times$ longer than the c-latency [16]. Given the approximately multiplicative role of network round-trip times (RTTs) when bandwidth is not the main bottleneck, eliminating inflation in Internet RTTs can potentially translate to up to 3-4$\times$ speedup, even *without* any protocol changes. Further, as protocol stack improvements get closer to their ideal efficiency of one RTT for small amounts of data, the RTT becomes the singular network bottleneck. Similarly, for well-designed applications dependent on persistent connectivity between two fixed locations, such as gaming, *nothing* other than resolving this 3-4$\times$ "infrastructural inefficiency" can improve latency substantially.

Thus, beyond the networking research community's focus on protocol efficiency, reducing the Internet infrastructure's latency inflation is the next frontier in research on latency. While academic research has typically treated infrastructural latency inflation as an unresolvable given, we argue that this is a high-value opportunity, and is much more tractable than may be evident at first.

What are the root causes of the Internet's infrastructural inefficiency, and how do we ameliorate them? Large latencies are partly explained by poor use of existing fiber infrastructure: two communicating sites often use a longer, indirect route because their service providers do not peer over the shortest fiber connectivity between their locations. We find, nevertheless, that even latency-optimal use of *all* known fiber conduits, computed via shortest paths in the InterTubes dataset [34], would leave us 1.98$\times$ away from c-latency [17]. This gap stems from the speed of light in fiber being $\sim\frac{2}{3}c$,

---

* Equal contribution.   † Now at Google.

and the unavoidable circuitousness of fiber routes due to topographic and economic constraints of buried conduits.

We thus explore the design of **cISP**, an Internet Service Provider that provides nearly speed-of-light latency by exploiting wireless electromagnetic transmissions, which can be realized with point-to-point microwave antennas mounted on towers. This approach holds promise for overcoming both the aforementioned shortcomings fundamental to today's fiber-based networks: the transmission speed in air is essentially equal to c, and the richness of existing tower infrastructure makes more direct paths possible. Nevertheless, it also presents several new challenges, including:

- overcoming numerous practical constraints, such as tower availability, line-of-sight requirements, and the impact of weather on performance;
- coping with limited wireless bandwidth;
- solving a large-scale cost-optimal network design problem, which is NP-hard; and
- addressing switching and queuing delays, which are more prominent with the smaller propagation delays.

To meet these challenges, we propose a hybrid design that augments the Internet's fiber connectivity with nearly straight-line wireless links. These low-latency links are used judiciously where they provide the maximum latency benefit, and only for the high-impact but small proportion, in terms of bytes, of Internet traffic that is latency-sensitive. We design a simple heuristic that achieves near-optimal results for the network design problem. Our approach is flexible and enables network design for a variety of deployment scenarios; in particular, we show that cISP's design for interconnecting large population centers in the contiguous U.S. and Europe can achieve mean latencies as low as $1.05 \times c$-latency at a cost of under \$1 per gigabyte (GB). We show through simulation that such networks can be operated at high utilization without excessive queuing.

To address the practical concerns, we use fine-grained geographic data and the relevant physical constraints to determine where the needed wireless connectivity would be feasible to deploy, and assess our design under a variety of scenarios with respect to budget, tower height and availability, antenna range, and traffic matrices. We also use a year's worth of meteorological data to assess the network's performance during weather disturbances, showing that most of cISP's latency benefits remain intact throughout the year. Our weather simulation and an animation showing how the hybrid network evolves from mostly-fiber to mostly-wireless with increasing budget are available online; see [25] and [26].

But is it feasible to use microwave hardware for low latency in practice? To answer this question, we rented virtual machines in the CME data center in Chicago and the Equinix data center in New Jersey, and, on Saturdays, were given access at these data centers to one of the fastest microwave networks spanning the Chicago – New Jersey algorithmic trading corri-

dor. Experiments conducted on this network show that it successfully operates at a speed extremely close to the speed of light, and that losses can be effectively handled by extremely lightweight forward error correction (FEC). We complement these findings by analyzing real trading data, revealing the minimum latency between the data centers and showing that the network is available in varied weather conditions.

Finally, we explore the application-level benefits for Web browsing and gaming, and present estimates showing that the utility of cISP vastly exceeds its cost, even for web sites already using CDNs to reduce latency.

2   TECHNOLOGY BACKGROUND

At the highest level, our approach involves using free-space communication between transmitters mounted at a suitable height, e.g., using dedicated towers or existing buildings, and separated from each other by at most a certain limiting distance. Network links longer than this range require a series of such transmitters. Typically, even after accounting for terrain, such a network link can be built close to the shortest path on the Earth's surface between the two end points. Further, the speed of light in air is essentially the same as that in vacuum, c. These properties make our approach attractive for the design of (nearly) c-latency networks.

Technology choices.   Several physical layer technologies are amenable for use in our design, including free-space optics (FSO), microwave (MW), and millimeter wave (MMW). At present, we believe MW provides the best combination of range, resilience, throughput, and cost. Future advances in any of these technologies, however, can be easily rolled into our design, and can only improve our cost-benefit analysis.

While hollow fiber [31] could, in the future, also provide c-latency, it would still suffer from the circuitousness of today's fiber conduits. Low Earth orbit satellite networks, as are being currently deployed, could also help, although they currently incur substantially higher latency than cISP (§9).

Switching latency.   While long-haul MW networks have existed since the 1940s [10], their use in high-frequency trading starting within the last 10 years [55] has driven innovation in radios so that each MW retransmission only takes a few μs. Thus, even wide-area links with many retransmissions incur negligible switching latency. As an example, the HFT industry operates a MW relay between Chicago and New Jersey comprising ≈ 20 line-of-sight links that operates within 1% of c-latency end-to-end at the *application* layer [58].

Packet loss.   Loss occurs for several reasons, including weather disruption and intermittent multi-path fading, especially over bodies of water. In §5.1, using a year's worth of weather data, we analyze the impact of diverting traffic to alternate (fiber or MW) routes during inclement weather. Our active experiments on a microwave network also show that losses experienced could be handled with lightweight forward error correction (FEC).

Spectrum and licensing.   We propose the use of MW com-

munication in the 6-18 GHz frequency range. These frequencies are not very crowded, and licensing is generally not very competitive, except at 6 GHz in cities, and along certain routes, like the above mentioned HFT corridor. The licenses are given on a first-come, first-served basis, recorded in a public database, they protect against the deployment of other links that would interfere with licensed links.

Line-of-sight & range. Successive MW towers need line-of-sight visibility, accounting for the Earth's curvature, terrain, trees, buildings and other obstructions, and atmospheric refraction. Attenuation also limits range. A maximum range of around 100 km is practicable, but we show results with maximum allowed range varying between 60-100 km (§5.2).

Bandwidth. Between any two towers, using very efficient encoding (256 QAM or higher), wide frequency channels, and radio multiplexing, a data rate of about 1 Gbps is achievable [45]. This bandwidth is vastly smaller than for fiber, and necessitates a hybrid design using fiber and MW.

Geographic coverage. Connecting individual homes directly to such a MW network would be cost-prohibitive. To maximize cost-efficiency, we focus on long-haul connectivity, with the last mile being traditional fiber. At short distances, fiber's circuitousness and refraction are small overheads.

Cost model. We rely on cost estimates in recent work [55] and based on our conversations with industry participants involved in equipment manufacturing and link provisioning. The cost of installing a bidirectional MW link, on existing towers, is approximately $75K ($150K) for 500 Mbps (1 Gbps) bandwidth. The average cost for building a new tower is $100K, with wide variation by terrain and across cities and rural areas. Any additional towers needed to augment bandwidth for particular links incur this "new tower" cost. The operational costs comprise several elements, including management and personnel, but the dominant operational expense, by far, is tower rent: $25 − 50K per year per tower. We estimate cost per GB by amortizing the sum of building costs and operational costs over 5 years.

Note that the deployment and operational costs can vary substantially based on the deployment model. For example, imagine that a company like American Tower [7], which has a substantial tower presence across the US (see Fig. 14 in Appendix D), deploys cISP. In such a scenario, not only would the cost of bandwidth augmentation be negligible, but also the cost of maintaining the towers would be drastically reduced. We consider both conservative and optimistic deployment models and conduct an in-depth cost-analysis in this work.

3 cISP DESIGN

At an abstract level, given the tower and fiber infrastructure, a set of n sites (e.g., cities, data centers) to interconnect, and a traffic model between them, we want to select a set of tower-level connections that minimizes network-wide latency while adhering to a budget and the constraints outlined in §2. Our approach comprises the following three broad steps.

1. Identifying a set of links that are likely to be useful by determining, for each pair of sites (s, d), the best feasible tower-level connectivity, if s and d were to be directly connected by a series of towers.

2. Building all $O(n^2)$ direct links, connecting each site to every other, would be prohibitively expensive. Thus, a subset of site-to-site links, together with existing fiber conduits, form our network. Choosing the appropriate subset is the key algorithmic problem.

3. Provisioning capacity beyond 1 Gbps along any link involves building additional tower-level links, e.g., by identifying and using links that are also nearly shortest paths, but were omitted in step 1 above.

Step 1: feasible hops. We first use line-of-sight and range constraints to decide which tower pairs can be connected. Achievable tower-to-tower hop length is limited primarily by the Earth's curvature, which can be treated as a "bulge" of height h_{Earth}. MW hops must clear this curvature and any obstructions in an ellipsoidal region between the sender and the receiver antennae known as the *Fresnel zone*, which has width h_{Fres}. At the midpoint of a hop of length D, using a MW frequency f, we have the following.

$$h_{\text{Fres}} \simeq 8.7\,m \left(\frac{D}{1\,\text{km}} \right)^{1/2} \left(\frac{f}{1\,\text{GHz}} \right)^{-1/2} \qquad (1)$$

$$h_{\text{Earth}} \simeq \frac{1\,m}{50\,K} \left(\frac{D}{1\,\text{km}} \right)^2 \qquad (2)$$

In Eq. 2, K accounts for atmospheric refraction [62]. Towers should clear the sum of these heights and any other obstructions. In favorable weather, and with adequately large dish antennae, ranges of up to $D \approx 100$ km are achievable with high availability, provided such line-of-sight clearance [79]. As a specific example, the FCC licensing database [28] indicates that McKay Brothers, LLC (a financial industry provider) operated a $D = 96$ km hop from Chicago, IL (lat. 41.88°, lon. -87.62°) to Galien, MI (lat. 41.81°, lon. -86.47°) as part of a 1183 km MW relay. This example shows that multipath interference issues (associated in this case with a traversal over Lake Michigan) are not an impediment to hop viability.

We assess hop feasibility between each pair of towers by using terrain data made available by NASA [66], which includes buildings and ground clutter, and effectively incorporates the height of the tree canopy.[1] We also require a fully clear Fresnel zone, and adopt $K = 1.3$ and $f = 11$ GHz in the above formulae. The hop engineering routines performing these calculations have been tested in practice: specifically, we have previously used them to design line-of-sight networks, at least 4 of which are now deployed, including ultra-low latency

---

[1]This NASA data set combines data from the Shuttle Radar Topography Mission (SRTM) [66] and the National Elevation Database (NED) [88], and typically yields acceptably small error (~ 2 m) against reference, high-accuracy LIDAR measurements.

routes between data centers hosting financial market matching engines. The methodology routinely provided correct clearance assessments when the physical paths were flashed (confirming line-of-sight with an on-site visit, e.g., [33]). It is relatively rare that the hop feasibility assessment is inaccurate; if a problem arises, it is most likely that the locations themselves are not available to rent. For this reason, in §5.2, we explore relaxations of our tower rental assumptions.

After identifying feasible tower-to-tower hops, for each pair of sites, we find the shortest path through a graph containing these hops, which we call a *link*. In line with observations from the tower data around major population centers, we assume each site itself hosts enough towers to use as the starting point for connectivity from that site to many others.

Step 2: topology design. We need to select a subset of site-to-site links to form a nationwide network that minimizes latency, given a limited budget to spend on links. The Steiner-tree problem [41] can be easily reduced to this problem, thereby establishing hardness. Standard approximation algorithms, like linear program relaxation and rounding, yield sub-optimal solutions, which although provably within constant factors of optimal, are insufficient in practice for this setting. Unfortunately, as we show in Appendix A, solving an Integer Linear program "unsplittable flow" formulation is intractable at the scales of interest. We thus propose two heuristics, the combination of which overcomes the scalability challenge, without substantially deteriorating solution quality.

The first observation we make is that the ILP formulation considers some flow variables that will never take non-zero values, allowing us to eliminate them and any resulting null constraints. For instance, if between two end points, a candidate microwave path is of higher latency than a fiber path (which we can always use, at negligible-in-comparison expense), then it will never carry any flow between these two end points. Similar observations apply to individual "distant, off-path" fiber and MW links. This simple observation substantially reduces the problem size. Note that standard network design problems do not typically have this structure available. This is entirely due to the hybrid design using fiber, which is assumed to be cheap, where available. We benefit, in this case, from having an "oracle" that tells us *a priori* when certain flow assignments are "obviously bad" and will not be useful. Further, carefully defined, such constraints preserve optimality; this part of our solution is not an approximation.

Second, we use a fast greedy heuristic to prune out MW links that are unlikely to be chosen. The heuristic operates using a larger budget ($2\times$ in our implementation) than we are ultimately allowed. In each iteration, we add to the solution the MW site-to-site link that decreases average stretch the most, continuing until the total cost reaches the inflated budget; the chosen links are candidates given to the ILP. Intuitively, the other links are uninteresting – they are unlikely to be picked in the final optimization even when a substantially larger budget is available, and so are not presented as options
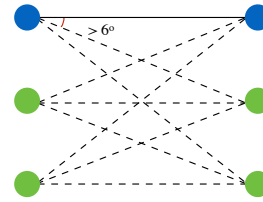
Fig. 1: k^2 *bandwidth with* $O(k)$ *new towers.*

to the ILP. This approach does not provide any guarantees, but we find that on small problem sizes, where the exact ILP can also be evaluated, it obtains the optimal solution.

Step 3: capacity augmentation. In many scenarios, some links require more capacity than a single MW connection. For short distances, this is a non-issue: the MW link can simply be replaced by fiber without a large impact on the network's latency. However, for longer distances, this is not acceptable.

One approach to resolving this problem is simply to build multiple parallel MW links, over multiple series of towers. While tower siting is often a challenging practical problem, with individual sites valued by the HFT industry at as much as \$14 million [59], in the cISP context there is a much larger "tolerance" than in HFT, where firms compete for fractions of microseconds. For a 500 km long cISP link, the midpoint diverging 10 km from the geodesic would increase latency by a negligible 0.2%. Thus, the problem of tower siting is substantially simpler. Also, in many cases, tower infrastructure is dense enough already to allow multiple parallel links. For instance, the HFT industry operates nearly 20 parallel networks in the New York-Chicago corridor [55].

We can also employ a simple trick to enhance the effectiveness of parallel series of towers, as shown in Fig. 1. Instead of k parallel series of towers providing merely a $k\times$ bandwidth improvement, connecting multiple antennae on each tower to other towers, we can obtain a $k^2\times$ improvement. Using antennae with overlapping frequencies requires an angular separation of $6°$ [62], as shown in Fig. 1. Again, the stretch caused by the resulting gap between parallel series of towers is small. For a tower-tower hop distance of 100 km, the minimum distance between two parallel towers should be $100 \cdot \tan(6°) = 10.6$ km, which, as noted above, has a small effect on end-to-end latency for long links.

This approach implies that for site-to-site bandwidths under 1 Gbps, we need just one series of towers; for bandwidths between 1-4 Gbps, we need 2 series; for 4-9 Gbps, 3; etc. While tower siting circumstances are often unique, we are aided by two observations: (a) there is substantial redundancy in existing tower infrastructure, and we can often find existing towers for parallel connections (see Fig. 3b and the related text in §4); and (b) when new towers are needed, there is substantial tolerance in where they are sited, as noted above. Bandwidth may potentially be increased even further through spatial diversity techniques, whereby multiple antennae are placed appropriately on the same tower such that they can adaptively cancel

interference by multiple transmission streams within the same frequency channel [89].

4 A cISP FOR THE UNITED STATES

We now apply the framework above for a concrete instantiation: designing a cISP for the U.S. mainland. To assess line-of-sight connectivity between existing towers, we use fine-grained data on tower infrastructure, buildings, terrain, and tree canopy. The fiber conduit data is available from past work [34].

Defining the sites and traffic model: To maximize utility while keeping costs low, we connect only the 200 most populous cities in the contiguous United States. In addition, we coalesce suburbs and cities within 50 km of each other, ending up with 120 population centers. (Henceforth, when we refer to "cities", we refer to these population centers.) Based on population data for 2010 [20], we calculate that 85% of the US population lives within 100 km of these 120 cities. For the traffic matrix, we use demands between city pairs that are proportional to their population product.

Step 1: Which city-city links are feasible? We use existing towers listed in FCC's Antenna Structure Registration [39] and databases from American Tower, Crown Castle, and several other tower companies for which we were able to download data. We cull these rather large databases of MW towers to a subset of 12,080 towers as follows: Towers from rental companies are typically suitable for use. From the FCC database, we only use towers over 100 m height. When tower-density exceeds 50 towers per 0.5° square grid cell, we randomly sample towers. (Using all towers could only improve our results, but increases compute time.)

Evaluating link feasibility across tower pairs within range of each other using the aforementioned NASA data [66], we find 261,019 tower-tower hops that satisfy line-of-sight constraints. We find that each city itself has large numbers of suitable towers in its vicinity. We run a shortest path computation on a graph comprising the cities and towers and city-tower and tower-tower hops to find the shortest city-city MW links. This yields both the cost (i.e., number of towers) and latency (i.e., distance along the chosen series of towers) for each city-city link.

For fiber distances, we compute the shortest paths over the InterTubes [34] dataset on US fiber conduits.

Step 2: What subset of links should we build? We use the Gurobi solver [42] to solve our topology design problem. As detailed in Appendix A: (a) both the exact ILP and an LP relaxation approach are too computationally inefficient, while our cISP design heuristic is able to solve the problem at the full scale; and (b) at small scales, where we can also run the exact ILP, our heuristic yields the optimal result.

Fig. 2 shows an example network. Designed with a budget of 3,000 towers and maximum hop length of 100 Km, its average latency is 1.05× c-latency. Fig. 3a shows the reduction of the network's stretch with increases in budget for maximum
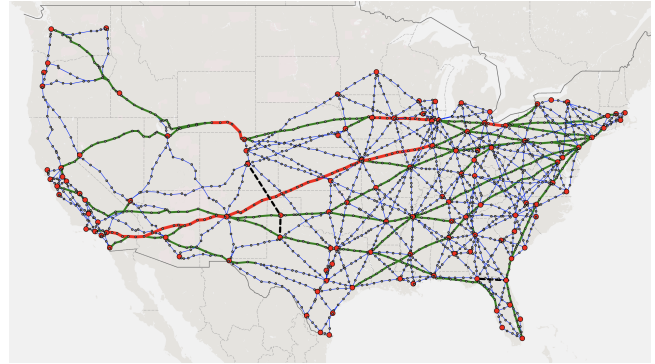
Fig. 2: *A 100 Gbps,* 1.05× *stretch network across 120 cities in the US. Blue links (thin) need no additional towers. Green (thicker) and red links (thickest) need 1 and 2 series of additional towers respectively. Black dashed links are fiber.*

hop lengths of 70 and 100 Km. Given the similarities with 70 and 100 Km, hereon, we only present results for the latter. An animation, showing how the network structure evolves from mostly-fiber to mostly-MW as the budget increases, is available online [26].

Step 3: Augmenting capacity. We produce a target aggregate demand (i.e., the sum of all site-site traffic demands) by scaling our traffic matrix. Then, each tower-tower MW hop that would be over-utilized (given shortest-path routing and the 1 Gbps capacity from §2) is augmented with additional towers at each end, as described in §3. Fig. 2's topology, when provisioned for an aggregate throughput of 100 Gbps, has 1,660 tower-tower hops that use only already-built towers seen in tower databases, while 552 hops need one additional new tower at each end, and 86 hops need 2 additional towers at each end. Using the cost model described in §2, we find that the cost per GB for this topology, with latency within 1.05× and 100 Gbps throughput, is $0.81. For some context, this is ∼ 10× the cost per GB for content delivery networks [64].

Provisioning even more bandwidth would require more new towers. For 1 Tbps, some tower-tower hops would need as many as 8 additional towers at each end. This is not infeasible — latency would not be inflated excessively, and towers could be found or built. In fact, for the long red link in the map in Fig. 2, which spans 2,700 km from Illinois to California, we find that the *longest* of these 8 additional series of towers would be only 5% longer than the shortest MW path, incurring a stretch of 1.07, instead of 1.02.

We can extend this argument even further: for the same Illinois to California link, we compute tower-disjoint shortest paths, i.e., after finding the shortest path, we remove all towers used by it, find the next-shortest tower-path, etc. In this process, we use only existing towers from our databases, and adhere to the same link feasibility constraints. Fig. 3b shows that stretch increases gradually as we keep eliminating towers; nevertheless, even after 20 such iterations, stretch is much smaller (1.15) than with the existing fiber conduit
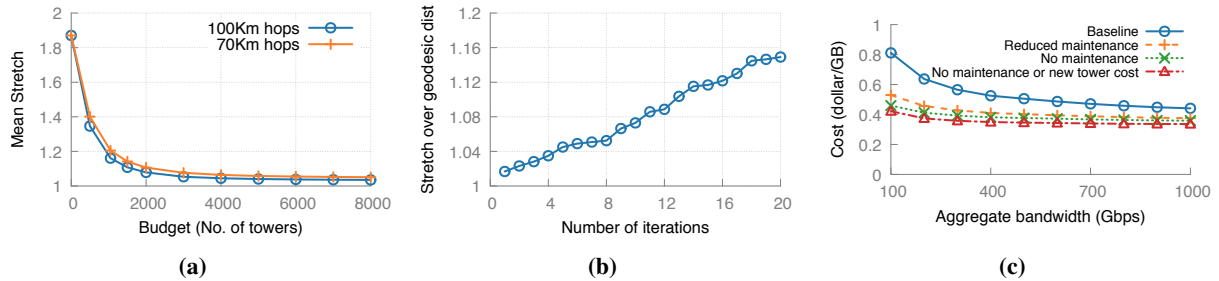
Fig. 3: *(a) Network stretch reduces as we add more MW towers. (b) Stretch for 20 shortest tower-disjoint purely MW paths along the long red IL-CA link in Fig. 2. (c) Cost per GB for the city-city traffic model decreases with increasing aggregate throughput.*
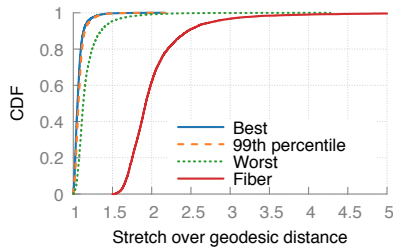

Fig. 4: *Stretch across all city-pairs over a year of weather. The 99th-percentile stretch is comparable to the best stretch.*

(1.75). Note that this route runs through the Rocky mountains and other areas of low tower density. Thus, in accounting for the cost of bandwidth augmentation entirely using the (higher) cost estimates for building new towers, we are substantially overestimating the expense.

There is also another reason our costs are over-estimates: at sufficiently high bandwidth, there is a better option than building many parallel long-distance MW links: one could use the same number of towers to construct a single line of towers with shorter tower-tower distances. This can make shorter-range, but higher-bandwidth technologies like MMW or free-space optics, more cost-effective.

Despite the above two factors, we use parallel MW towers, with all the required additional towers accounted for as new towers, to provide conservative cost-estimates as aggregate bandwidth increases in Fig. 3c.

Routing, queuing, and traffic models. We show in Appendix B that: (a) routing that incurs small (under 10%) latency inflation compared to shortest paths can drive the network at virtually zero loss and minimal queuing delay even at high utilization; and (b) packet pacing addresses the problem of edge links having higher line rates than cISP links. We also discuss evidence for per-MW-hop latency overheads being small enough to ignore. Further, in Appendix C, we show that besides the population-product model, cISP can also be tailored for inter data center traffic, data center to edge traffic, and various combinations of these.

Alternative deployment models. The deployment model and analysis have been conservative in assuming high maintenance and tower installation costs for the provider. What if an incumbent tower company like American Tower [7] deployed cISP? (Fig. 14 in the Appendix shows that American

Tower's existing deployment broadly covers areas where our network design of Fig. 2 requires towers.) Besides reduced tower installation costs, maintenance would also be significantly reduced due to the obligation to maintain towers for customers anyway. We evaluated several scenarios of this type, as shown in Fig. 3c. While the solid line represents the baseline deployment model discussed in §2, the dashed lines represent models with reduced maintenance cost ($10K per tower per year), no maintenance cost, and no maintenance or new tower cost (only antenna cost). A network with 3,000 towers offering 100 Gbps bandwidth and 1.05 stretch, built by a company like American Tower, could cost as little as $0.42/GB, thus reducing the baseline cost by almost 50%.

Finally, we note that cISP could be deployed in other geographies besides the US. As discussed in Appendix C.3, we could design a cISP for Europe offering a stretch of 1.04 (vs. 1.05 for the U.S.) with a budget of ∼3k towers.

5 PRACTICAL CHALLENGES

Deploying cISP would involve several practical challenges beyond network design and routing, which we now address.

5.1 Impairments due to weather

We use standard equations from MW engineering [48] to calculate signal attenuation due to precipitation. We assume hardware characteristics of a standard low-latency MW radio: an 8-foot dish with a gain of 46.5 dBi at 11 GHz [29, 74, 75]. While antenna gain is determined by the hardware, transmit power and receive power thresholds also depend on the modulation scheme (256 QAM). Following ITU models [48], at ∼11 GHz, precipitation is likely to be the dominant source of attenuation. While the physical layer could trade link bandwidth for higher resilience to weather, we treat the impact of precipitation in a binary manner: if attenuation exceeds a threshold that would degrade bandwidth, we conservatively consider a link to have failed.

We assume that when a link fails, traffic is shifted to the shortest available route, which may use any combination of MW and fiber. The high precipitation that causes failures is easy to predict, especially on the timescale of minutes. Thus, even slow, centralized management would suffice to anticipate failures and reroute accordingly.

We use NASA's precipitation data [65] to determine which links are down when, and what the impact of such failures
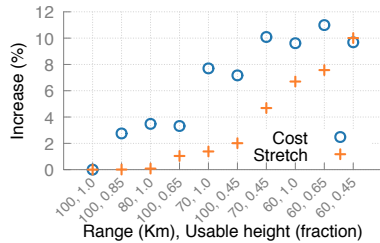
Fig. 5: *As constraints on tower space and range become tighter, the network becomes more expensive, and stretch increases.*

is on the network's latency. For each day over a period of a year (July 2015 - June 2016), we select a 30-minute interval uniformly at random, and identify the links that would fail during it. We then evaluate the latency for each pair of cities end-to-end for each interval. Fig. 4 shows that 99th-percentile latencies are nearly the same as the best fair-weather latencies. In terms of the median across city-pairs, even the worst latencies over the year are 1.7 times lower than those over fiber. Large increases in latency due to weather typically occur only between nearby city-pairs, the fiber route to which runs through a farther-away city, e.g., in Texas, Austin and Killeen fall back to a fiber route through Fort Worth. A more sophisticated analysis allowing dynamic link bandwidth adjustment rather than binary failures can only improve these numbers. Thus, even under significantly adverse weather, most of the latency advantage of cISP remains intact.

We have also created an animated visualization of the network's latency evolving over a year's weather [25].

5.2 Tower height and availability

Our initial design assumed a MW hop to be feasible if it spans a distance of 100 km or less, and satisfies line-of-sight constraints using the tops of the towers. In practice, however, a tower chosen for a route might not have a free spot for a new antenna at the necessary height, especially at the top, where structural concerns for large parabolic antennae are greatest, and where access and maintenance can be problematic. Further, for smaller antennas, insufficient gain margins can decrease the 100 km maximum range. Hence, we evaluate cost and latency of the network with hop-level restrictions modeling these effects.

We test the impact of restricting usable height on towers to three levels, as a fraction of tower height: 0.85, 0.65, and 0.45. Testing for line-of-sight visibility with these restrictions eliminates more towers than using tower tops. We also vary the maximum range, which can necessitate the use of a larger number of towers, thus increasing the cost and potentially making some city-pairs infeasible to connect using MW.

We assess the *percentage increase* in cost and stretch values compared to the baseline values with 100 km range and using the tower tops, i.e., height fraction = 1. Fig. 5 shows the results for different combinations of the range and antenna-height constraints, sorted by lowest to highest stretch. The

maximum increase in cost is 11% (with the absolute cost per GB under these constraints being $0.90), while the maximum increase in stretch is 10% (with the absolute stretch compared to the geodesic being 1.16). Thus, even substantial potential problems with mounting antennas do not change our overall conclusions about the viability of cISP.

In our experience designing MW routes, assessments like the ones in this work have yielded accurate estimates of the latency and the number of tower-tower hops that will ultimately be used to connect two sites. The precise set of towers often differs based on real-world constraints, particularly tower unavailability for structural and rental-related reasons. Thus, while accurate in terms of cost and latency, this work does not provide fully engineered routes. In practice, to improve accuracy in preparation for building a MW route, we assign an acquisition probability to each tower in a swath connecting the sites, which depends on a number of factors (e.g., tower type, ownership, and location). Further, for towers that can be acquired, we use a uniform distribution to model height at which space for antennas is available. With this probabilistic model, we compute thousands of candidate MW paths between site pairs, with refinements as acquisitions and height availabilities are confirmed. We make available in video form [24] an example of such refinement.

5.3 Integration into the Internet

We next discuss potential problems cISP may face in terms of integration into the present Internet ecosystem.

Low-hanging fruit: The easiest deployment scenarios involve one entity operating a significant network backbone:

- A CDN could use cISP to carry "back-office" traffic between its locations and content origins, which often supports latency-sensitive user-facing interactions [73]. While the strategies of moving content closer to end users and speeding up the network are orthogonal, on cache misses and when serving uncacheable content, only speeding up the network improves performance.

- Content-providers like Google and Facebook can use cISP to carry lateny-sensitive traffic – such WAN designs already accommodate distinctions between such traffic and background traffic [47, 50].

- Purpose-built networks such as for gaming [40] can easily use cISP between their edge locations and servers.

All of these are interesting and economically viable use cases with minimal deployment barriers, and each *alone* may justify a design like cISP. For instance, while it is tempting to dismiss gaming as a niche, it is a large and growing market: the Steam gaming platform claims 20+ million players worldwide [85]. At a 10 Kbps rate per player [27], this aggregates to 27 Gbps – enough to make cISP viable in this setting. (We present cost-benefit estimates, including for gaming, in §8.)

User-facing deployment: Access ISPs may use cISP as an additional provider, and incorporate a low-latency service

into their broadband plans.[2] Utilizing cISP in this manner can help ISPs to provide and meet the requirements of demanding Service Level Agreements, the case for which was made in recent work [14]. ISPs may use heuristics to classify latency-sensitive traffic and transit it using cISP. Alternatively, software at the user-side may make more informed decisions about which traffic should use the fast-path exposed by the ISP. While this would require significant user-side changes, note that many of today's applications already manage multimodal WiFi and cellular connectivity.

6 EMPIRICAL RESULTS

To evaluate the characteristics of long-haul microwave links, we have conducted experiments over one of the most popular nearly-speed-of-light networks deployed in the high-frequency trading corridor between Chicago and New Jersey. We describe these experiments and their results below. The HFT niche is partially characterized by a "winner-takes-all" dynamic which requires these networks to operate at the bleeding edge of low latency. Hence, it is important to quantify the usefulness of these networks in serving more generic low-latency applications on the Internet, which have less-strict latency requirements than HFT, but higher availability and lower packet loss demands.

6.1 Active measurements

We conducted active measurements over the microwave link between the Chicago Mercantile Exchange (CME) data center and the Equinix data center in Secaucus, New Jersey, operated by one of the fastest MW networks in the corridor. On weekdays, when the Chicago and New York markets are open, the link carries financial information critical to high-frequency trading that triggers trades worth billions of dollars. The networks are optimized for low latency, with microseconds of advantage [13] providing a significant edge to customers.

We ran experiments for ∼7 hours every Saturday for 11 weeks between Nov. 2019, and Oct. 2020 from one host each located in the CME and Equinix data centers. The microwave link was provided to us without any Forward Error Correction (FEC), thus being exposed to all errors and bit flips expected in radio transmission. We observe that the link behavior tends to be in one of two states: losses are either very low (normal) or very high (degraded). Out of a total of 72 hours of measurements, there are 12 hours during which the link is degraded due to weather, and 4 hours during which it is down due to maintenance or other issues. Note that because there is no FEC at all, very small bit error rates (BER) degrade the link. Also, in our trading data analysis (§6.2), we see that microwave networks stay up in worse weather conditions than these 12 hours. FEC is needed in packet headers to correct for bit errors, which we could not implement as we did not have access to routers on the network.

---

[2] While large last-mile latencies can overshadow cISP's low latency, this is an entirely orthogonal problem, on which significant progress is being made – 5G prototypes are already showing off sub-millisecond latencies [46].

6.1.1 RTT and bandwidth

The geodesic distance between the CME and Equinix data centers is 1139.5 km. The c-latency for a round-trip, then, is 7.6 ms. In our experiments over 11 weeks, we always observe a round-trip time of 7.7 ms for 32-byte packets, i.e., within 1.5% of c-latency. The RTT goes up to 7.9 ms for 1,499-byte packets because of the limited bandwidth available on the link (or more specifically, the slice of it provided to us).

The 0.1 ms increase in transmission delay as packet size increases by 1,467 bytes gives a bandwidth estimate of 120 Mbps. Our UDP measurements and TCP measurements, in the best case, also give us a bandwidth of 120 Mbps. It is hard for TCP to sustain throughput at this rate in the absence of any FEC because of transmission losses. While the operator did not divulge the exact link capacity, it is likely that our network access was capacity-capped. Hence, these measurements only provide a lower bound on the link bandwidth.

6.1.2 Loss and FEC

In plain TCP (iperf) and ICMP (ping) probes, we observe high loss rates: typically around 3% to 5% for 32-byte packets. The packet loss rate increases sharply as packet size increases because more bits can potentially be corrupted in transmission. Without FEC, a link with loss rate this high is clearly unsuitable for web traffic [91]. Whether FEC can bring the loss rate down to an acceptable level (say, 0.1%) at reasonable latency and bandwidth overhead depends on two factors: 1. the Bit Error Rate (BER), and 2. the typical length of error bursts, i.e., how many consecutive bits are corrupted in an error burst. We elaborate on these factors below.

First, we derive the underlying BER from observed ping packet loss. For a ping packet of s bytes, a successful response is observed when both the echo request and reply packets are delivered to the respective hosts without any errors. To estimate the BER b_{err}, we first assume that bit errors are uniform and random. Then, for packet loss rate p_{loss}, we get:

$$b_{err} = 1 - (1 - p_{loss})^{1/(2 \times 8 \times s)}$$

For initial validation of this model, with the possibly unjustified assumption of random and uniform errors, we calculate b_{err} from observed p_{loss} for $s = 1,499$ for the 7 hours of measurements on Feb. 15th, 2020. Then, we use the calculated b_{err} to predict p_{loss} for $s = 396$ on the same day. We compare the predicted and observed values in Fig. 6a. While the observed and predicted loss rates for $s = 396$ largely agree, there are some disagreements, e.g., at 12:30, which can be explained by the fact that the observations for $s = 1,499$ and $s = 396$ are separated in time by 60 seconds. The underlying BER might change during this interval. For Feb. 15th, the median, 95th percentile, and maximum BER we calculate are 3.6×10^{-5}, 8.2×10^{-5}, and 3.6×10^{-4} respectively.

For a target packet loss rate of 0.1% for packets of size 1,500 bytes, the BER needs to be 4.17×10^{-8} or lower. Extremely lightweight FEC codes, such as Reed-Solomon (255,
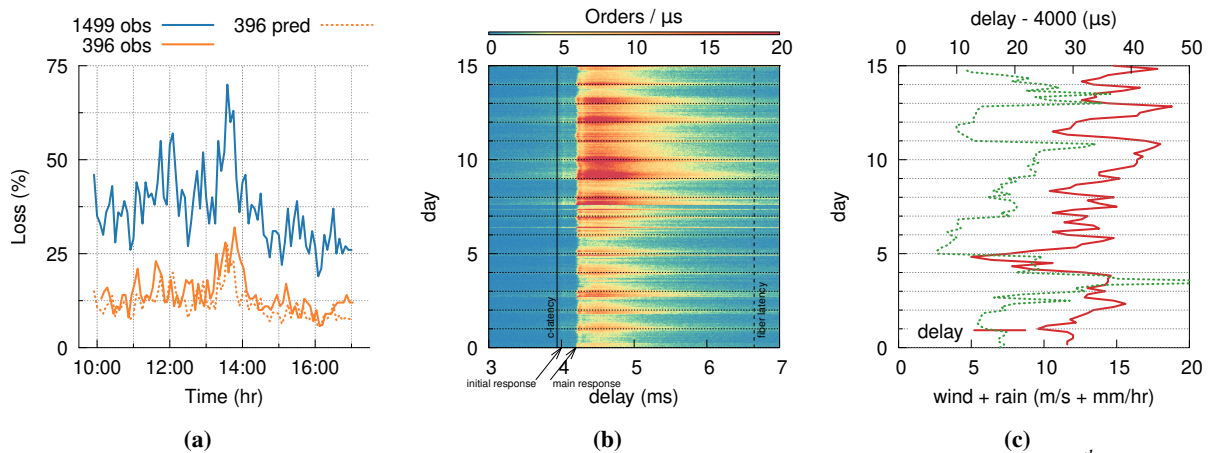
Fig. 6: *(a) Predicting loss rate of 396 byte packets from observed loss rates of 1,499 byte packets on Feb 15th, 2020. Analyzing trading data: (b) Heat map of order book events at delay between Chicago and New Jersey. Response delay never exceeds 4.3 ms; (c) A coarse weather signal (max wind speed + max rainfall) is correlated with the observed transmission delay.*

239) can correct from BER of 10^{-4} to 10^{-12} with a bit rate overhead of only 7% [76]. If performed over 255 byte blocks, a 1,500 byte packet can be encoded in 7 blocks with a total redundancy overhead of 112 bytes. At 120 Mbps bandwidth, this incurs a latency penalty of only 7.5 μs. This FEC scheme would break down, however, if errors occurred in bursts of around 8 bytes or more. Now we discuss the earlier assumption of error bursts being short and uniformly distributed.

To analyze bit errors, we sent two sets of UDP probes over the link: the first set consists of 60 byte packets sent at 35 packets per second (slow), and the second consists of 60 byte packets sent at 200,000 packets per second (fast). The slow set characterizes link behavior with no congestion/bandwidth related losses, whereas the fast set provides statistical significance to rare bit flip events. In contrast to ping losses, losses in this experiment are observed through packet captures rather than at the application layer, so a corruption of, e.g., the UDP destination port would not register a loss. For the slow set, we observe a packet loss rate of 0.8%, whereas for the fast set we observe a loss rate of 2.04%.

In the UDP fast set a packet has 4 bytes of payload, 8 bytes of UDP header, 20 bytes of IP header, 14 bytes of Ethernet header, and 14 bytes of padding. A total of 1.6 billion packets were sent, out of which 2.66 million were received on the other end with at least one of the following fields corrupted: source port, destination port, UDP header length field, and payload. We calculate the Hamming distance between the received value and the expected value of the corrupted fields. As Table. 7a shows, there appears to be a linear relationship between field size and number of corruptions, and over 99% of all corruptions consist of 2 bit flips or less. Also, if we extrapolate the errors we observe in these 4 fields to the rest of the 60 byte packet, the expected loss rate due to corruptions in the Ethernet and IP headers and padding matches that observed in the UDP slow set. The other 1.24% packets lost can thus be explained by congestion/bandwidth issues.

6.2 Trading data analysis

To characterize the latency and up-time of the full range of microwave links deployed in the Chicago-New Jersey corridor, we analyze trading data from the Chicago Mercantile Exchange (CME) in Chicago, Illinois, and the CBOE Options Exchange in Secaucus, New Jersey. Information about trades happening at the CME travels over microwave paths and triggers activity at the CBOE [13]. The time difference between stimulus events at the CME and the response at the CBOE represents the network latency between the two exchanges. Laughlin et al. have also used this methodology to estimate latency between financial markets [55].

We obtained tick data from CME and CBOE for three weeks of Mar. 2019. The tick data consists of microsecond precision timestamps for events at both ends. Both markets are open simultaneously for 6.5 hours every weekday, which means that we have 97.5 hours of relevant tick data. For each trade executed at the CME at timestamp t, we count the number of order book events at the CBOE at timestamps $t + i$ where $i \in [3000, 7000]$ μs. Fig. 6b plots a heat map of the number of orders per μs for each 10 μs bin in the tick data. The y-axis time is in intervals of 15 minutes. Analysis of the data shows that the main response delay, which reflects the network latency between CME and CBOE, does not exceed 4.3 ms for any 15-minute interval. The lowest fiber latency between the two exchanges is 6.65 ms [60]. This shows that some microwave networks were up through every 15-minute interval over the 3-week period.

In addition to the main response at 4.2 ms, Fig. 6b has a smaller initial response at 4.0 ms. The CME tick data reveals that internal trading algorithms and strategies produce a second stimulus at CME 200 μs after the initial stimulus. The main response in Fig. 6b is triggered by that second stimulus.

We consider the delay between the second stimulus at CME and the main response at CBOE as transmission delay. We calculate the transmission delay for every 1-hour interval

Field	#corruptions	#bits	1 bit flip	2 bit flips
src port	873,165	16	84%	15%
dst port	864,955	16	82%	17%
length	914,528	16	85%	14%
payload	1,734,539	32	84%	15%

(a)
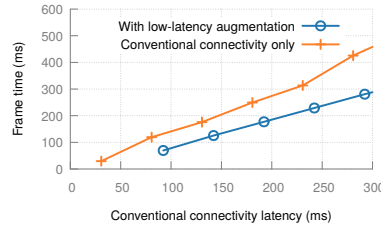
(b) **(c)**

Fig. 7: *(a) Corruptions observed in the UDP fast set. (b) A substantial reduction in frame time can be obtained by the use of a parallel low-latency augmentation to the present Internet. (c) Mean web page load time (PLT) improvement for each heuristic and its portion of traffic delivered on cISP. PLT can improve substantially by only offloading a small portion of traffic to cISP.*

in the tick data. Fig. 6c plots the moving average of transmission delay over 2 hours. We use the hourly wind speed estimate [30] and rainfall data [22] in the regions through which the MW corridor passes as a coarse weather signal. For each hour, we pick the maximum wind speed and maximum rainfall observed at a granularity of ~ 10 km along the geodesic between the end points. Fig. 6c plots wind speed + rainfall /2, and shows that there is some correlation. The Pearson correlation coefficient between wind and delay is 0.24, while that between rain and delay is 0.16. Sources of noise in this correlation include the noise inherent in the trading data itself, and issues that may affect transmission delay, such as infrastructure damage or operational downtime. Note that days 3 and 14 have more severe rain and wind than the 12 hours during which the link was degraded in our active measurements (§6.1).

Conclusions: From the active measurements, we conclude that for our MW path, (1) round-trip latency is less than 1.5% inflated over *c*-latency, (2) bandwidth is at least 120 Mbps, (3) error bursts are very short and roughly uniformly distributed under normal link conditions, and (4) errors can be brought down to acceptable levels with extremely lightweight FEC incurring minimal latency and bandwidth overhead.

From the trading data analysis, we conclude that (1) for the 97.5-hour period, some MW networks, spanning more than 1,000 km, were always up without any significant degradation in latency, and (2) weather events such as high wind speeds and rainfall are correlated with increases in transmission delay by tens of microseconds. This increase may stem from one or more of the following: (a) longer end-to-end paths being picked, (b) shorter tower-to-tower hops leading to higher switching delay, and (c) the network responding to weather changes by ramping up FEC.

7 A FEW POTENTIAL APPLICATIONS

Several applications require low latency over the wide area-network. Applications focused on user interactivity, such as augmented and virtual reality, tele-presence and tele-surgery, musical collaboration over long-distances, etc., can all benefit from low-latency connectivity. Likewise, less user-centric applications, such as real-time bidding for Web page adver-

tisements [8] and block propagation in blockchains, would also benefit. While it is beyond the scope of this paper to analyze this in detail, we assess, in simplified environments, the improvements cISP could achieve for two applications.

7.1 Online gaming

We discuss cISP's benefits for both models of online gaming: **thin-client** (where a client essentially streams everything in real-time from a server) and **fat-client** (where the client has the game installed, performs computations, etc., and only relies on the server for updates on the global game state).

Fat-clients are dominant today, and are easy to tackle: communication is almost entirely composed of latency-sensitive player actions and game-state changes, and is low-volume, typically a few Kbps per client for popular games [27]. It can all be transferred over the low-latency network, reducing latency by 3-4× compared to today's Internet.

Thin-client gaming is still in its infancy, as it depends heavily on the network, with data rates in Mbps. We explore the potential of a speculative approach: the server speculates on the game state and sends data for multiple scenarios in advance over fiber, then on the low-latency network, issues messages indicating which scenario occurred. Such speculation has already shown success for rich games like "Doom 3" [56].

We use a toy thin-client for a multi-player Pacman variant to explore the latency benefit. Our rudimentary implementation speculates on all 4 movement directions possible as user input. In line with the online-gaming literature, we measure "frame-time," which "corresponds to the delay between a user's input and the observed output" [56]. We evaluate frame-time as latency over conventional connectivity increases (emulated by adding latency in software), and for a low-latency network always incurring 1/3 of the latency of the corresponding conventional network.

As Fig. 7b shows, the speculative approach enabled by the low-latency network augmentation reduces frame-time. This comparison would improve further if non-network overheads from processing and rendering in our naive implementation were smaller. We do not use any heavy graphics on which to evaluate the additional bandwidth overhead on fiber, but even in the sophisticated scenarios examined by prior work [56], this bandwidth overhead can be contained to 2-4.5×.

7.2 Web Browsing

We evaluate the potential impact of cISP's latency improvement on Web page load times (PLTs) (based on the *onLoad* event [71]) using Mahimahi [68] with the addition of content delivery network (CDN) caching. Our emulation supports two levels of the CDN cache hierarchy. The client's request first reaches the edge server. If it is a cache miss, the request will be forwarded to the parent server. In case there is another miss, it will be forwarded to the origin server. This setup thus allows variable request latency, where certain requests can experience more latency.

To realistically recreate the caching behavior, our experiments leverage the Akamai pragma header [3] which is typically used for debugging purposes. We select web pages where at least 75% of the HTTP requests[3], performed when loading a page, are served by Akamai. Overall, we found 27 landing pages and 140 associated internal pages from the Hispar list [9] match this criterion. We record each page's content and the network latency for each (edge) server that a client contacted when loading a page. This recording process is conducted from three different vantage points at three different times. For the CDN server-to-server latency, we estimate the latency by geolocating the IP addresses of the CDNs and origin servers provided by the pragma header[4]. We then replay each page with unmodified network latencies (as a baseline) and with latencies reduced to $0.33\times$ of their original values (as a cISP). No bandwidth limitations are imposed.

Fig. 7c shows the results. Compared to the baseline, a 66% reduction in latencies (all-cISP) results in a mean 42% PLT decrease for both landing and internal pages (an absolute decrease of 600 ms and 651 ms). This PLT reduction is less than the 66% reduction in RTT because loading a Web page also involves significant non-network activity.

If cISP is used only to deliver the CDN's server-to-server (i.e., back-office) traffic, our experiment (backhaul-cISP) suggests that PLT can be improved by 23.7% and 28.5% (331 ms and 447 ms) on landing and internal pages by only sending 13.4% and 22.3% of the overall web-browsing traffic on cISP. Internal pages get better improvement and send a higher proportion of traffic because they experience more cache misses (31.9%) compared to landing pages (13.3%).

While Web-browsing traffic comprises only a small fraction of total Internet traffic[5], we can further reduce the load by carrying only *latency-sensitive* traffic on cISP. Hence, we extend Mahimahi to enable *selective* manipulation of RTTs in the replay, such that some traffic sees lower RTTs than other traffic. We test two heuristics under this setup. First, we try a simple heuristic that only sends uplink traffic to cISP (uplink-cISP). This approach yields a mean PLT improvement of 21.5% (319 ms) by sending only 9.7% of the web-browsing traffic over cISP. Second, we adopt a more advanced PKT-State heuristic [77] (packet-cISP) to distinguish the latency-sensitive traffic (e.g., TCP SYN/ACK packets and small data packets) from the bandwidth-intensive traffic (e.g., data packets). By offloading the latency-sensitive traffic to cISP, we can get a mean PLT improvement of 28.2% (417 ms) by only offloading 10.2% of the traffic.

8 COST-BENEFIT AND MARKET ANALYSIS

Does cISP's value justify its cost? For three important use cases, we present quantitative lower-bound estimates of cISP's value per GB. cISP would also need enough aggregate demand across one or more use cases to support its total deployment cost, so we estimate market size of each use case.

Web search. *Value per GB:* Putting together Google's quantification of the impact of latency in search [18], their estimated search revenue restricted to the US [63], their search volume [84], estimated data transferred per search[6], and estimated cost per search [53], we estimate that speeding up page load times for 12 Gbps of their US search traffic by only 200 ms (400 ms) would yield an additional yearly profit of $87 ($177) million. This translates to an added value of $1.84 ($3.74) per GB. *Market size:* At 12 Gbps of traffic, Google's search traffic is a nontrivial fraction ($> 10\%$) of a cISP provisioned to provide $\sim$100 Gbps, but to make cISP viable, it would have to be augmented with other use cases.

E-commerce. *Value per GB:* Using Amazon.com's estimates of number of visits, pages fetched per visit, fraction of US traffic [80], and page size, we arrive at an estimated 480 PB of US traffic per year. Using their US sales [32] and profit margin of 5.5% [61] gives an estimated $16.3 billion in profits per year. Estimates for the effect of PLT on conversion rate vary from 1% [57] to 2.4% (on desktop) and 7% (on mobile) per 100 ms of additional latency [5]. Thus, saving 200 ms by sending only 10% of the data over cISP (§7.2), translates to a value of $6.8-$47.5 per GB, which is much higher than the $0.81 per GB cost of cISP traffic. *Market size:* 10% of 480 PB of Amazon e-commerce annual US traffic translates to 12 Gbps of cISP traffic. But the current (2020) e-Commerce market size of $861 billion [32] (compared to Amazon's $\sim$\$296 billion) proportionately translates to a cISP traffic demand of 35 Gbps. Given the high value per GB, this use case alone could make a 100 Gbps cISP profitable.

Gaming. *Value per GB:* Online gamers often pay for "accelerated VPNs", which promise to lower network latency. Such services cost $4-$10 per client per month [1, 11, 72]. Full-time gaming at 8 hours a day at a 10 Kbps rate (as in §5.3) translates to 1.08 GB / month. Thus, if cISP were priced like a cheap accelerated VPN service at $4 / mo, this would translate to a value of at least $3.7 / GB. A less aggressive model than "full-time gaming" would only improve cISP's value. Another indicator of latency's value in gaming is the

---

[3]We assume requests not served by Akamai are served by the edge server.
[4]We geolocate each server, and compute server-to-server c-latency from distance. Then, we estimate baseline latency as $3\times$ c-latency.
[5]Cisco's 2018 estimate puts "Web/Data traffic" at 13% [23] including non-latency sensitive traffic like software updates and some file transfers.

[6]From Firefox desktop's network tools; mobile responses may be smaller.

market for gaming monitors with high screen-refresh rates: the 6-10 ms of latency advantage is valued at over $50 by many gamers, estimated from the pricing of monitors which are exactly the same except in terms of refresh rate [6]. *Market size:* There are more than 350 million [83] Fortnite gamers worldwide. Assuming 20% of the gamers are in the US, each with a demand of 10 Kbps, translates to 700+ Gbps of cISP demand. Even for games with smaller user bases like PUBG (70 million) and Call of Duty Warzone (100 million), cISP demands are high enough to sustain a nationwide network.

Summary. The value per GB obtained from cISP's latency reduction in the above cases – $1.84-$3.74, $6.52-$45.63, and over $3.70 – exceeds its cost estimate of $\leq$ $0.81 per GB, and even leaves room for substantial over-provisioning. Total addressable market demand could greatly exceed a 100 Gbps cISP for the case of gaming, and for web-based use cases could be sufficient to support the infrastructure.

This simplified analysis omits many factors. Not all users would be paying for the infrastructure on day 1, so an incremental roll-out for a smaller set of customers would be important. Also, there are many other applications that can benefit from cISP. CDNs routinely use overlay routing to cut latency for dynamic, non-cacheable content, for which edge replication is difficult or ineffective [4]. Upcoming application areas like virtual and augmented reality can only make the case stronger for cISP. We expect cISP's most valuable impact to be in breaking new ground on user interactivity, as explored in some depth in prior work [16].

9   RELATED WORK

Networking research has made significant progress in measuring latency, as well as improving it through transport, routing, and application-layer changes. However, the underlying infrastructural latency has received little attention and has been assumed to be a given. This work proposes a speed-of-light ISP, demonstrating that improvements are indeed possible.

There are several ongoing Internet infrastructure efforts, including X moonshot factory's project Taara [90], Facebook connectivity's Magma [36], Rural Access [37], Terragraph [38], and the satellite Internet push by Starlink [81], Kuiper [54], Telesat [86], and others. Project Taara consists of networks under deployment in India and Africa, based on free-space optics, and described as "Expanding global access to fast, affordable internet with beams of light". While Facebook's Magma and Rural Access aim to extend connectivity to rural areas by offering a software, hardware, business model, and policy framework, Terragraph aims to extend last-mile connectivity to poorly connected urban and suburbans areas by leveraging short millimeter-wave hops. Free-space networks of this type will likely become more commonplace in the future, and these works are further evidence that many of the concerns with line-of-sight networking can indeed be addressed with careful planning. Further, cISP's design approach is flexible enough to incorporate a variety of media

(fiber, MW, MMW, free-space optics, etc.) as the technology landscape changes.

"New Space" satellite networks: While low-Earth orbit (LEO) satellite networks can reduce long-distance latency [12, 44, 52], current deployments are more targeted at last-mile connectivity than long haul [15]. Starlink recently claimed to offer last-mile round-trip latency of 31 ms [82], more than $3.8\times$ the latency estimated in prior simulations [12], showing that the service is not yet latency optimized.

Despite the apparent differences in objectives — long haul latency for cISP and last-mile connectivity for LEO networks — it is useful to **coarsely** assess how the costs may compare. Starlink, for example, offers uncapped connectivity at $99/month [78]. At an average household consumption of 273.5 GB [35], this translates to $0.36/GB[7]. For cISP, if an incumbent like American Tower were to deploy it, the cost could be as low as $0.33/GB, as shown in Fig. 3c. Thus, a network with costs comparable to cISP (in a per-bit sense; cISP is more than an order of magnitude cheaper in absolute cost, and has commensurately lower bandwidth) is concurrently being deployed, albeit with different goals.

To the best of our knowledge, the only efforts primarily focused on wide-area latency reduction through infrastructural improvements are in niches, such as the point-to-point links for financial markets [55], and isolated submarine cable projects aimed at shortening specific Internet routes [67, 69].

10   CONCLUSION

A speed-of-light Internet not only promises significant benefits for present-day applications, but also opens the door to new possibilities, such as *eliminating the perception of wait time* in our interactions over the Internet [16]. We thus present a design approach for building wide-area networks that operate nearly at *c*-latency. Our solution integrates line-of-sight wireless networking with the Internet's fiber infrastructure to achieve both low latency and high bandwidth.

A speed-of-light Internet has not always been clearly viable. The enabling technology of low-latency multi-hop microwave networks was spurred on by HFT only within the last 10 years, and even then it has not been a priori obvious that the challenges of relatively high loss and low bandwidth could be overcome to leverage such links for an Internet backbone. More importantly, the Internet has become increasingly latency-limited due to increasing bandwidths and greater use of interactive applications. Thus, we believe we have reached an exciting point in time when greatly reducing the Internet's infrastructural latency is not only tractable, but surprisingly cost-effective and impactful for applications.

---

[7]Starlink is currently in beta testing, and profit margins are unclear. It is difficult to do a tighter cost analysis for Starlink without more information.

REFERENCES

[1] AAA Internet Publishing, Inc. WTFast. https://www.wtfast.com/en/. [Online; accessed 11-March-2021].

[2] Akamai. Akamai "10for10". https://www.akamai.com/us/en/multimedia/documents/brochure/akamai-10for10-brochure.pdf, July 2015. [Online; accessed 11-March-2021].

[3] Akamai. Using Akamai Pragma headers to Investigate or Troubleshoot Akamai Content Delivery. https://community.akamai.com/customers/s/article/Using-Akamai-Pragma-headers-to-investigate-or-troubleshoot-Akamai-content-delivery?language=en_US, 2015. [Online; accessed 11-March-2021].

[4] Akamai. SureRoute. https://developer.akamai.com/learn/Optimization/SureRoute.html, 2017. [Online; accessed 11-March-2021].

[5] Akamai. The State of Online Retail Performance. https://www.akamai.com/uk/en/multimedia/documents/report/akamai-state-of-online-retail-performance-spring-2017.pdf, 2017. [Online; accessed 11-March-2021].

[6] amazon.com. ASUS VG248QE Gaming Monitor. https://goo.gl/gnFnPv, 2018. [Online; accessed 11-March-2021].

[7] American Tower Global Wireless Solutions. https://www.americantower.com/us/, 2004. [Online; accessed 11-March-2021].

[8] Waqar Aqeel, Debopam Bhattacherjee, Balakrishnan Chandrasekaran, P. Brighten Godfrey, Gregory Laughlin, Bruce Maggs, and Ankit Singla. Untangling header bidding lore: Some myths, some truths, and some hope. In *Passive and Active Measurement*, 2020.

[9] Waqar Aqeel, Balakrishnan Chandrasekaran, Bruce Maggs, and Anja Feldmann. On landing and internal pages: The strange case of Jekyll and Hyde in Internet measurement. In *ACM IMC*, 2020.

[10] AT&T Corporation. AT&T Long Lines Routes March 1960. http://long-lines.net/places-routes/maps/MW6003.html, 2003. [Online; accessed 11-March-2021].

[11] Battleping. Info on our lower ping service. http://www.battleping.com/info.php, 2010. [Online; accessed 11-March-2021].

[12] Debopam Bhattacherjee, Waqar Aqeel, Ilker Nadi Bozkurt, Anthony Aguirre, Balakrishnan Chandrasekaran, P Godfrey, Gregory Laughlin, Bruce Maggs, and Ankit Singla. Gearing up for the 21st century space race. In *ACM HotNets*, 2018.

[13] Debopam Bhattacherjee, Waqar Aqeel, Gregory Laughlin, Bruce M. Maggs, and Ankit Singla. A bird's eye view of the world's fastest networks. In *ACM IMC*, 2020.

[14] Zachary S. Bischof, Fabián E. Bustamante, and Rade Stanojevic. The Utility Argument - Making a Case for Broadband SLAs. In *PAM*, 2017.

[15] Bloomberg. Musk targets telecom for next disruption with Starlink Internet. https://tinyurl.com/wejrv37c, 2021. [Online; accessed 11-March-2021].

[16] Ilker Nadi Bozkurt, Anthony Aguirre, Balakrishnan Chandrasekaran, Brighten Godfrey, Gregory Laughlin, Bruce M. Maggs, and Ankit Singla. Why Is the Internet so Slow?! In *PAM*, 2017.

[17] Ilker Nadi Bozkurt, Waqar Aqeel, Debopam Bhattacherjee, Balakrishnan Chandrasekaran, Philip Brighten Godfrey, Gregory Laughlin, Bruce M. Maggs, and Ankit Singla. Dissecting latency in the Internet's fiber infrastructure, 2018. arXiv:1811.10737.

[18] Jake Brutlag. Speed Matters for Google Web Search. http://goo.gl/vJq1lx, 2009. [Online; accessed 11-March-2021].

[19] Gustavo Carneiro, Pedro Fortuna, and Manuel Ricardo. FlowMonitor: A Network Monitoring Framework for the Network Simulator 3 (NS-3). In *Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools*, VALUETOOLS '09, 2009.

[20] Center for International Earth Science Information Network (CIESIN), Columbia University; United Nations Food and Agriculture Programme (FAO); and Centro Internacional de Agricultura Tropical (CIAT). Gridded Population of the World: Future Estimates (GPWFE). http://sedac.ciesin.columbia.edu/gpw, 2005. [Online; accessed 11-March-2021].

[21] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F. Wenisch. The mystery machine: End-to-end performance analysis of large-scale internet services. In *USENIX OSDI*, 2014.

[22] CHRS at UC Irvine. PERSIANN-CCS. https://chrsdata.eng.uci.edu/, 2017. [Online; accessed 11-March-2021].

[23] Cisco. Cisco Visual Networking Index: Forecast and Methodology. https://www.reinvention.be/webhdfs/v1/docs/complete-white-paper-c11-481360.pdf, 2017. [Online; accessed 11-March-2021].

[24] cISP authors. MW path refining. https://goo.gl/LwYB5Z. [Online; accessed 11-March-2021].

[25] cISP authors. Impact of rainfall on cISP for a period of 1 year. https://tinyurl.com/a8szcukz, 2021. [Online; accessed 11-March-2021].

[26] cISP authors. The MW+fiber hybrid network evolves with budget. https://tinyurl.com/3vakxccm, 2021. [Online; accessed 11-March-2021].

[27] Mark Claypool, David LaPoint, and Josh Winslow. Network analysis of Counter-Strike and Starcraft. In *IEEE Performance, Computing, and Communications Conference*, 2003.

[28] Federal Communications Commission. Universal Licensing System. http://wireless2.fcc.gov/UlsApp/UlsSearch/searchLicense.jsp. [Online; accessed 11-March-2021].

[29] CommScope. HSX8-107-D3A. https://objects.eanixter.com/PD354739.PDF, 2012. [Online; accessed 28-July-2021].

[30] Copernicus by ECMWF. ERA5 hourly data on single levels from 1979 to present. https://cds.climate.copernicus.eu/cdsapp#!/dataset/reanalysis-era5-single-levels?tab=overview, 2018. [Online; accessed 11-March-2021].

[31] DARPA. Novel Hollow-Core Optical Fiber to Enable High-Power Military Sensors. http://www.darpa.mil/news-events/2013-07-17, 2013. [Online; accessed 11-March-2021].

[32] Digital Commerce 360. US ecommerce grows 44.0% in 2020. https://www.digitalcommerce360.com/article/us-ecommerce-sales/, 2021. [Online; accessed 28-July-2021].

[33] DragonWave-X. Services & Support / Pre Deployment / Line of Sight. https://www.dragonwavex.com/services/pre-deployment/line-sight, 2021. [Online; accessed 11-March-2021].

[34] Ramakrishnan Durairajan, Paul Barford, Joel Sommers, and Walter Willinger. InterTubes: A study of the US long-haul fiber-optic infrastructure. In *ACM SIGCOMM*, 2015.

[35] Joan Engebretson. Broadband Data Usage Report: Internet-only Homes Use Almost Twice as Much Data as Bundled Homes. https://www.telecompetitor.com/broadband-data-usage-report-internet-only-homes-use-almost-twice-as-much-data-as-bundled-homes/, 2019. [Online; accessed 11-March-2021].

[36] Facebook connectivity. Magma. https://connectivity.fb.com/magma/, 2021. [Online; accessed 11-March-2021].

[37] Facebook connectivity. Rural Access. https://connectivity.fb.com/rural-access/, 2021. [Online; accessed 11-March-2021].

[38] Facebook connectivity. Terragraph. https://connectivity.fb.com/terragraph/, 2021. [Online; accessed 11-March-2021].

[39] Federal Communications Commission. Antenna Structure Registration Database. https://www.fcc.gov/antenna-structure-registration, 2018. [Online; accessed 11-March-2021].

[40] Riot Games. Fixing the Internet for real-time applications. https://goo.gl/SEoxW2, 2016. [Online; accessed 11-March-2021].

[41] M. R. Garey and D. S. Johnson. The rectilinear Steiner tree problem is NP-complete. *SIAM Journal on Applied Mathematics*, 32(4):826–834, 1977.

[42] Inc. Gurobi Optimization. Gurobi optimizer reference manual, 2016.

[43] Nikola Gvozdiev, Stefano Vissicchio, Brad Karp, and Mark Handley. Low-latency routing on mesh-like backbones. *ACM HotNets*, 2017.

[44] Mark Handley. Delay is not an option: Low latency routing in space. In *ACM HotNets*, 2018.

[45] Jonas Hansryd and Jonas Edstam. Microwave capacity evolution. *Ericsson review*, 1:22–27, 2011.

[46] Devindra Hardawar. Samsung proves why 5G is necessary with a robot arm. https://goo.gl/3gZTn8, 2016. [Online; accessed 11-March-2021].

[47] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven WAN. In *ACM SIGCOMM*, 2013.

[48] ITU. Specific attenuation model for rain for use in prediction methods. http://www.itu.int/dms_pubrec/itu-r/rec/p/R-REC-P.838-3-200503-I!!PDF-E.pdf, 2005. [Online; accessed 11-March-2021].

[49] Ixia. Measuring Latency in Equity Transactions. http://ixia.cabanday.com/products/_content/wp-measuring-latency.pdf, 2012. [Online; accessed 11-March-2021].

[50] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: experience with a globally-deployed software defined wan. In *ACM SIGCOMM*, 2013.

[51] Srikanth Kandula, Dina Katabi, Bruce Davie, and Anna Charny. Walking the tightrope: Responsive yet stable traffic engineering. In *ACM SIGCOMM*, 2005.

[52] Simon Kassing, Debopam Bhattacherjee, André Baptista Águas, Jens Eirik Saethre, and Ankit Singla. Exploring the "Internet from space" with Hypatia. In *ACM IMC*, 2020.

[53] Kevin Kelly. How much does one search cost? http://kk.org/thetechnium/how-much-does-o/, 2007. [Online; accessed 11-March-2021].

[54] Kuiper Systems LLC. Application of Kuiper Systems LLC for Authority to Launch and Operate a Non-Geostationary Satellite Orbit System in Ka-band Frequencies. https://licensing.fcc.gov/myibfs/download.do?attachment_key=1773885, 2019.

[55] Gregory Laughlin, Anthony Aguirre, and Joseph Grundfest. Information transmission between financial markets in Chicago and New York. *Financial Review*, 2014.

[56] Kyungmin Lee, David Chu, Eduardo Cuervo, Johannes Kopf, Yury Degtyarev, Sergey Grizan, Alec Wolman, and Jason Flinn. Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming. In *ACM MobiSys*, 2015.

[57] Greg Linden. Make Data Useful. https://slideplayer.com/slide/4203392/, 2006. [Online; accessed 11-March-2021].

[58] McKay Brothers LLC. Quincy Extreme Data Latencies. http://www.quincy-data.com/product-page/#latencies, 2017. [Online; accessed 11-March-2021].

[59] Brian Louis. Trading Fortunes Depend on a Mysterious Antenna in an Empty Field. https://goo.gl/82kzXd, 2017. [Online; accessed 11-March-2021].

[60] Donald MacKenzie. *Trading at the Speed of Light: How Ultrafast Algorithms Are Transforming Financial Markets*. Princeton University Press, 2021.

[61] Macrotrends LLC. Amazon Net Profit Margin 2006-2021. https://www.macrotrends.net/stocks/charts/AMZN/amazon/net-profit-margin, 2021. [Online; accessed 28-July-2021].

[62] Trevor Manning. *Microwave Radio Transmission Design Guide*. Artech House, 2009.

[63] Ginny Marvin. Report: Google earns 78% of $36.7B US search ad revenues, soon to be 80%. https://goo.gl/kp4L5X, 2017. [Online; accessed 11-March-2021].

[64] Microsoft Azure. Content Delivery Network pricing. https://azure.microsoft.com/en-us/pricing/details/cdn/, 2018. [Online; accessed 11-March-2021].

[65] NASA. Precipitation Processing System Data Ordering Interface for TRMM and GPM (STORM). https://storm.pps.eosdis.nasa.gov/storm/, 2015. [Online; accessed 11-March-2021].

[66] NASA Jet Propulsion Laboratory. U.S. Releases Enhanced Shuttle Land Elevation Data. https://www2.jpl.nasa.gov/srtm/, 2015. [Online; accessed 11-March-2021].

[67] NEC. SEA-US: Global Consortium to Build Cable System Connecting Indonesia, the Philippines, and the United States. https://tinyurl.com/ybj9nhp3, August 2014. [Online; accessed 11-March-2021].

[68] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. Mahimahi: Accurate record-and-replay for http. In *USENIX ATC*, 2015.

[69] A. Nordrum. Fiber optics for the far north [news]. *IEEE Spectrum*, 52(1):11–13, January 2015.

[70] ns-3 community. Network simulator ns-3. https://www.nsnam.org, 2011. [Online; accessed 11-March-2021].

[71] Jan Odvarko. Har 1.2 spec. http://www.softwareishard.com/blog/har-12-spec, 2007. [Online; accessed 11-March-2021].

[72] Pingzapper. Pingzapper Pricing. https://pingzapper.com/plans, 2018. [Online; accessed 11-March-2021].

[73] Enric Pujol, Philipp Richter, Balakrishnan Chandrasekaran, Georgios Smaragdakis, Anja Feldmann, Bruce M. Maggs, and Keung-Chi Ng. Back-office web traffic on the Internet. In *ACM IMC*, 2014.

[74] radiowaves. SHPD8-1011. https://www.radiowaves.com/getmedia/b1a7277f-fde0-4c05-a5fc-7c22c29c5b3a/HPD8-1011.aspx, 2018. [Online; accessed 28-July-2021].

[75] radiowaves. SPD8-11. https://www.radiowaves.com/getmedia/f942ec58-9999-4607-a165-fd4db4deef60/SPD8-11.aspx, 2018. [Online; accessed 28-July-2021].

[76] Eduard Sackinger. *Analysis and Design of Transimpedance Amplifiers for Optical Receivers*. John Wiley & Sons, 2017.

[77] William Sentosa, Balakrishnan Chandrasekaran, P. Brighten Godfrey, Haitham Hassanieh, Bruce Maggs, and Ankit Singla. Accelerating mobile applications with parallel high-bandwidth and low-latency channels. In *ACM HotMobile*, 2021.

[78] Michael Sheetz. SpaceX prices Starlink satellite internet service at $99 per month, according to e-mail. https://www.cnbc.com/2020/10/27/spacex-starlink-service-priced-at-99-a-month-public-beta-test-begins.html, 2020. [Online; accessed 11-March-2021].

[79] Shkilko, A. and Sokolov, K. Every Cloud Has a Silver Lining: Fast Trading, Microwave Connectivity and Trading Costs. https://ssrn.com/abstract=2848562, 2016. [Online; accessed 11-March-2021].

[80] SimilarWeb. Overview: amazon.com. https://www.similarweb.com/website/amazon.com/#overview, 2021. [Online; accessed 28-July-2021].

[81] SpaceX Starlink. https://www.spacex.com/webcast, 2017. [Online; accessed 11-March-2021].

[82] Starlink Services. Petition of Starlink Services, LLC for designation as an eligible telecommunications carrier. https://ecfsapi.fcc.gov/file/1020316268311/Starlink%20Services%20LLC%20Application%20for%20ETC%20Designation.pdf, 2021. [Online; accessed 11-March-2021].

[83] statista. Online gaming - statistics & facts. https://www.statista.com/topics/1551/online-gaming/, 2021. [Online; accessed 28-July-2021].

[84] Internet Live Stats. Google Search Statistics. https://www.internetlivestats.com/google-search-statistics/. [Online; accessed 11-March-2021].

[85] Steam. Steam & game stats, 2017. http://store.steampowered.com/stats/ [Online; accessed 11-March-2021].

[86] Telesat. Telesat: Global Satellite Operators. https://www.telesat.com/, 2020. [Online; accessed 11-March-2021].

[87] Unwired Labs. OpenCelliD Tower Database. https://opencellid.org/, 2018. [Online; accessed 11-March-2021].

[88] USGS. National Elevation Dataset (NED). https://www.usgs.gov/core-science-systems/national-geospatial-program/national-map. [Online; accessed 11-March-2021].

[89] J. H. Winters, J. Salz, and R. D. Gitlin. The impact of antenna diversity on the capacity of wireless communication systems. *IEEE Transactions on Communications*, 42(2/3/4):1740–1751, Feb/Mar/Apr 1994.

[90] X, the moonshot factory. Taara – Expanding global access to fast, affordable internet with beams of light. https://x.company/projects/taara/, 2018. [Online; accessed 11-March-2021].

[91] Xiufeng Xie, Xinyu Zhang, and Shilin Zhu. Accelerating mobile web loading using cellular link information. In *ACM MobiSys*, 2017.

A TOPOLOGY DESIGN

Picking a subset of site-to-site links to connect a set of cities involves solving a typical network design problem. The Steiner-tree problem [41] can be easily reduced to this problem, thereby establishing hardness. Standard approximation algorithms, like linear program relaxation and rounding, yield sub-optimal solutions, which although provably within constant factors of optimal, are insufficient in practice. We develop a simple heuristic, which, by exploiting features specific to our problem setting, obtains nearly optimal solutions.

Inputs: Our network design algorithm requires:

- A set of sites to be interconnected, $v_1, v_2, \ldots, v_n$.
- A traffic matrix H specifying the relative traffic volume $h_{ij} \in [0, 1]$ between each pair v_i and v_j.
- The geodesic distance d_{ij} between each v_i and v_j.
- The distance along the shortest, direct MW path between each pair, m_{ij}, as well as its cost, c_{ij}. This is part of the output of step 1.
- The optical fiber distance between each pair, o_{ij}, which we multiply by 1.5 to account for fiber's higher latency.
- A total budget B limiting the maximum number of bidirectional MW links that can be built.

Expected output: The algorithm must decide which direct MW links to pick, i.e., assign values to the corresponding binary decision variables, x_{ij}, such that the total cost of the picked links fits the budget, i.e., $\sum_{ij} x_{ij} c_{ij} \leq B$. Our objective is to minimize, per unit traffic, the mean stretch, i.e., the ratio of latency to c-latency, where c-latency is the speed-of-light travel time between the source and destination of the traffic.

Problem formulation: Expressing such problems in an optimization framework is non-trivial: we need to express our objective in terms of shortest paths in a graph that will itself be the *result*. We use a formulation based on network flows.

Each pair of sites (v_s, v_t) exchanges h_{st} units of flow. To represent flow routing, for each potential link ℓ, we introduce a binary variable $f_{stij,m}$ which is 1 *iff* the $v_s \to v_t$ flow is carried over the microwave link $v_i \to v_j$, and a binary variable $f_{stij,o}$ which is 1 *iff* the same flow is carried over the optical link[8] $v_i \to v_j$. The objective function is:

$$min \sum_{s,t} \frac{h_{st}}{d_{st}} \sum_{i,j} (o_{i,j} f_{stij,o} + m_{i,j} f_{stij,m}) \qquad (3)$$

The h_{st} term achieves our goal of optimizing *per unit traffic*. The $\frac{1}{d_{st}}$ term achieves our goal of optimizing the *stretch*.

---

[8]A "link" between sites can use multiple physical layer hops, both for MW and fiber. The underlying multi-physical-hop distances are already captured by the inputs o_{ij} and m_{ij} so the optimization views it as a single link.

For brevity, we omit the constraints, which include: flow input and output at sources and sinks; flow conservation; total budget; and the requirement that only links that are built ($x_{ij} = 1$) may carry flow. All variables are binary, so flows are "unsplittable" (carried along a single path) and the overall problem is an integer linear program (ILP).

Note that we have decomposed the problem so that link capacity is *not* a constraint in this formulation: MW links will be built with sufficient capacity in step 3; fiber links are assumed to have plentiful bandwidth at negligible cost relative to MW costs. As a result, the objective function will guide the optimizer to direct each $v_i \rightarrow v_j$ flow along the shortest path of built links, which is the direct MW link $v_i \rightarrow v_j$ if it happens to be built, or otherwise, a path across some mix of one or more fiber and MW links.

ILP's limited scalability: The exact ILP is not scalable, which is the reason we use multiple heuristics, as discussed in §3. As we show in Fig. 8a, the exact ILP, without using our observations on the problem structure, is too computationally inefficient to scale to this scenario. We use subsets of all 120 cities to assess scalability, with the budget proportional to the number of cities in each test, with a budget of 6,000 towers at the largest scale. Even after 2 days of compute, the exact ILP was unable to obtain a result for sets of cities larger than 50. In contrast, our cISP design heuristic is able to solve the problem at the full scale. Second, as Fig. 8b shows, at small scales, where we can also run the exact ILP, our heuristic yields the optimal result. We also tested a linear program rounding approach, but even the naive LP relaxation followed by rounding did not scale beyond 60 cities, and gave results worse than optimal.

B ROUTING & QUEUING

The HFT industry's point-to-point MW deployments demonstrate end-to-end application layer latencies within 1% of *c*-latency, after accounting for all delays in microwave radios, interfacing with switching equipment and servers, and application stacks. Such low latencies across point-to-point long-distance links place sharp focus on any latencies introduced at routers for switching, queuing, and transmission.

Internet routers can forward packets in a few tens of microseconds, and specialized hardware can hit $100\times$ smaller latencies [49]. Transmitting 1500 B frames at 1 Gbps takes 12 μs. Thus forwarding and transmission even across many long-distance links incur negligible latency. Longer routes and queuing delays, however, can have substantial impact.

To assess the impact of routing and queuing in cISP, we use ns-3 [70]. We use UDP traffic with a uniform packet size of 500 bytes. We use the built-in FlowMonitor [19] to measure delay and loss rate, and add a new monitoring module to track link-level utilization. All experiments simulate 100 Gbps of network traffic for one second of simulated time. An experiment takes approximately 10 hours to complete on a single core of a 3.1 GHz processor. Even achieving this running time

requires some compromises: we aggregate the bandwidth of parallel links and remove the individual tower hops to focus on network links between the routing sites.

Routing schemes: Besides ns-3's default shortest path routing, we implement two other schemes – throughput optimal routing, and routing that minimizes the maximum link utilization, a scheme commonly employed by ISPs [51].

Results: When the traffic and routing match the design target, i.e., the population-product traffic routed over shortest paths, we find that the network can be driven to high utilization (95%) with near-zero queuing and loss. Non-shortest-path routing schemes needlessly compromise on latency in such scenarios. (Plots for this easy scenario are omitted.)

We also test the network's behavior under deviations from the designed-for traffic model. We emulate scenarios where a city produces more or less traffic than expected by allowing, for each city, a "population perturbation" — each city's population is re-weighted by a factor drawn from the uniform distribution $U[1-\gamma, 1+\gamma]$ for a chosen $\gamma \in [0,1]$.

Fig. 9a and Fig. 9b show the results for $\gamma \in \{0.1, 0.3, 0.5\}$. Even for large perturbations, the mean delay does not increase by more than 0.1 ms and the loss rate is zero up to an aggregate load of 70% of the capacity designed for, even with just shortest path routing. Other routing schemes are indeed more resilient to higher load, achieving virtually zero loss and queuing delay even at high utilization, but at the cost of latency. For the tested topology, both the alternative routing schemes incur 10% higher latency on average (not shown in the plots). These results indicate there would be significant value in work that reduces the amount of over-provisioning required by making modest compromises on latency on some routes, e.g., as in [43].

Speed mismatch: The bandwidth disparity between the network core and edge for cISP may seem atypical, in the sense that in most settings, the core has higher bandwidth links compared to the edge, while in cISP, edge links (such as those at large data center end points) may often have much higher line rates when they feed their outgoing traffic into cISP. Thus, we also evaluate if this "speed mismatch" causes persistent congestion at cISP's ingresses.

We run ns-3 simulations with several sources (S_i) connected to a sink (*D*) through the same intermediate node (*M*). The *M-D* link rate is fixed at 100 Mbps. We then evaluate settings with every S_i-*M* link being either 100 Mbps or 10 Gbps. The former is the control, and the latter is the setting with a speed mismatch. *M* has an unbounded queue. Ten sources send 100 KB TCP flows (small, as is expected in cISP) to the sink, *D*. The arrival of these TCP flows follows a Poisson process, consuming on average 70% of the *I-D* link's bandwidth. Each simulation run lasts 10 s and we conduct 100 such runs. We test TCP both with and without pacing.

Fig. 10a shows that the median queue occupancy at *M* is higher without pacing, especially at the 95th percentile. However with pacing, queueing behavior is nearly the same.
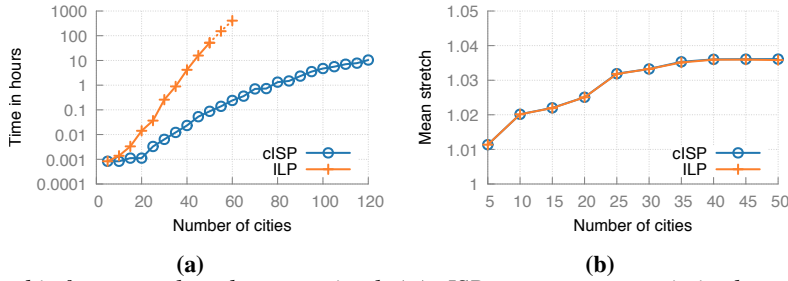
(a)   **(b)**

Fig. 8: *cISP's design method is fast-enough and near-optimal: (a) cISP generates an optimized topology within hours for* 120 *cities while the ILP does not yield a result even after* 2 *days for more than* 50 *cities. For the ILP, runtimes for* 50+ *cities are extrapolated by curve fitting. (b) The stretch achieved by cISP matches that of the ILP to two decimal places for instances that can be optimized by the ILP.*
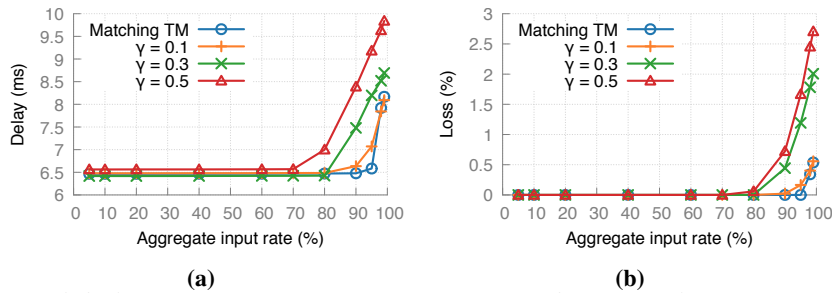

(a)   **(b)**

Fig. 9: *(a) Average delay and (b) loss rate remain consistent across perturbations of the city-city traffic model, except under heavy load.*

(a)

(b)
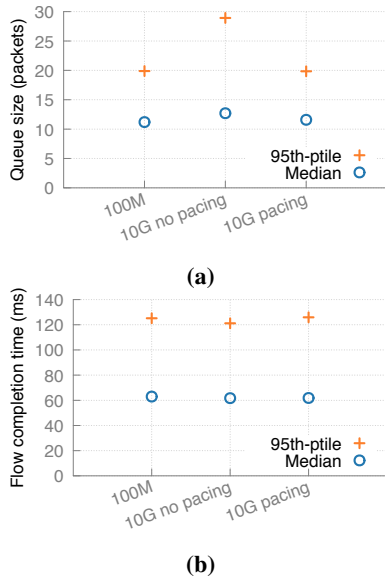
Fig. 10: *TCP pacing addresses the problem of capacity mismatch (a) by reducing persistent queuing (b) without affecting flow completion times.*

The median flow completion times (Fig. 10b) are unaffected both with and without pacing.

C   FURTHER DESIGN CONSIDERATIONS

C.1   Is the city-city traffic model special?

Ideally, we would be able to use wide-area traffic matrices from some ISP or content provider for modeling. In the absence of such data, we focus on showing that cISP can be tailored to vastly different deployment scenarios and their cor-

responding traffic models. Apart from the city-city population product model, we use (a) traffic between a provider's data centers; and (b) traffic between the cities and data centers.

An inter data center cISP: We use Google data centers as an example, considering all 6 publicly available US locations - Berkeley, SC; Council Bluffs, IA; Douglas County, GA; Lenoir, NC; Mayes County, OK; and The Dalles, OR. In the absence of known inter-data center traffic characteristics, we provision equal capacity between each DC-pair.

Data centers to the edge: We also model a scenario where data centers are to be connected to edge locations in cities. Each of the 120 cities connects to its closest Google data center, with traffic proportional to its population.

We show in Fig. 11 that using the same design approach as in §3, both of the above scenarios result in networks with lower cost than the city-city model. Thus, cISP can be tailored to a variety of use cases and traffic models.
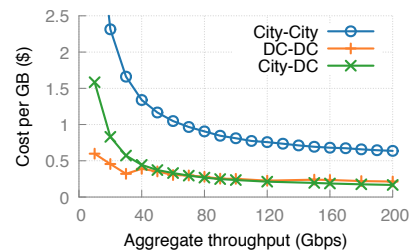
Fig. 11: *Cost per GB for different traffic models: the City-City model, discussed in the most detail, is the most expensive.*
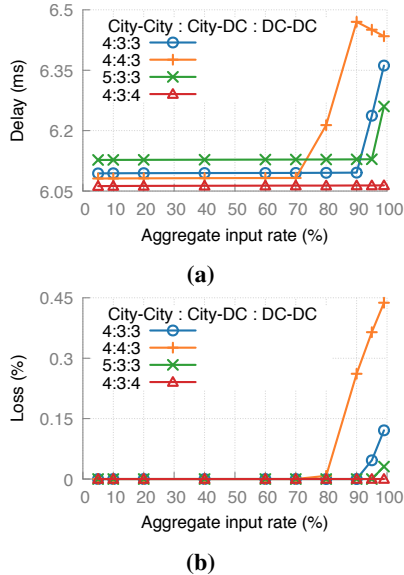
Fig. 12: *(a) Average delay and (b) loss rate remain consistent across deviations from the designed-for traffic mix, except under heavy load.*

C.2 Traffic model mismatches

A cISP may carry a mix of city-city, inter-DC, and DC-edge traffic. How does its performance degrade as the *proportion* of these traffic types departs from the design assumptions?

We design a cISP to carry an aggregate of 100 Gbps with a city-city : DC-edge : inter-DC traffic proportion of 4:3:3. Using ns-3 simulations similar to those in §B, we then test this network under several traffic mixes different from this designed-for mix — 5:3:3, 4:3:4, and 4:4:3.

Fig. 12a and Fig. 12b show that there is a difference of less than 0.05ms in mean delay across different combinations of traffic matrices up to an aggregate load of 70% of the design capacity. Similarly, loss remains nearly 0 until this load. The decrease in delay at high load (4:4:3 for $x > 90$ in Fig. 12b) is due to losses, which are likelier on longer, higher-delay paths.

Mean delay depends more on city-city traffic, as expected: city-city traffic requires a wider infrastructure footprint, and deviations from its design parameters have greater impact.

Thus, as discussed in §B, significant traffic model deviations can be absorbed using some over-provisioning, in line with current ISP practices.

C.3 Is the US geography special?

It is reasonable to ask: are the population distribution and geography of the U.S. especially amenable to this approach, or is it applicable more broadly? The availability of high-quality tower data and geographical information systems data for the U.S. enables a thorough analysis. While similar data is, unfortunately, not available to us for other geographies, we can approximately assess the design of a cISP in Europe using public, crowd-sourced data on cellular towers [87]. Lacking fiber conduit data, we assume that fiber distances between

Fig. 13: *A 100 Gbps $1.04\times$ stretch cISP across Europe. This network uses several fiber connections (dashed, black lines).*

cities are inflated over geodesic distance in the same way as in the US ($\sim 1.9\times$). Using our methodology in §3, we design a European cISP of similar geographical scale across cities with population more than 300k, targeting the same aggregate capacity and mean latency ($1.04\times$ here vs. $1.05\times$ for cISP-US). The cost of this design, shown in Fig. 13, is similar as well, with $\sim$3k towers. Note that the impact of Europe's higher population density is not seen here, because we explicitly design for the same aggregate throughput. One could, alternatively, normalize throughput per capita, and compare cost per capita, to obtain similar results.

Admittedly, there is not yet a known approach to bridging large transoceanic distances using MW, limiting our approach to large contiguous land masses that need to be interconnected with fiber. In the distant future, LEO satellite links, hollow-core fiber, or even towers on floating platforms may be of use for such connectivity.

D American Tower deployment
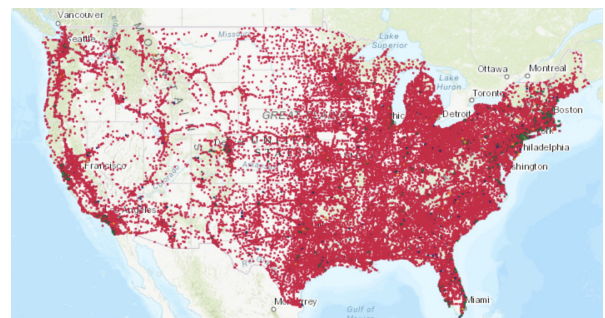
Fig. 14: *American tower deployment as per 5th March, 2021.*

American Tower [7] claims to have a presence at more than 42,000 tower sites across the US, as of 5th March 2021. Fig. 14 shows their current deployment. We could not access their database due to legal bindings.

Configanator: A Data-driven Approach to Improving CDN Performance.

Usama Naseer
Brown University

Theophilus A. Benson
Brown University

Abstract

The web serving protocol stack is constantly evolving to tackle the technological shifts in networking infrastructure and website complexity. As a result of this evolution, web servers can use a plethora of protocols and configuration parameters to address a variety of realistic network conditions. Yet, today, despite the significant diversity in end-user networks and devices, most content providers have adopted a "one-size-fits-all" approach to configuring the networking stack of their user-facing web servers (or at best employ moderate tuning).

In this paper, we demonstrate that the status quo results in sub-optimal performance and argue for a novel framework that extends existing CDN architectures to provide programmatic control over a web server's configuration parameters. We designed a data-driven framework, Configanator, that leverages data across connections to identify their network and device characteristics, and learn the optimal configuration parameters to improve end-user performance. We evaluate Configanator on five traces, including one from a global content provider, and evaluate the performance improvements for real users through two live deployments. Our results show that Configanator improves tail (p95) web performance by 32-67% across diverse websites and networks.

1 Introduction

Web page performance significantly impacts the revenue of content distribution networks (CDNs) (e.g., Facebook, Akamai, or Google), with studies showing that a 100ms decrease in page load times (PLT) can lead to 8% better conversion rate for retail sites [14, 30]. Yet, uniformly improving web performance is becoming increasingly challenging due to the growing disparity in the network conditions (e.g. bandwidth, RTT) [3, 36, 120, 135] and end-user devices [93, 94, 108, 134]. To address this disparity and improve the quality of experience (QoE), the networking community is constantly developing new protocols and configuration parameters for web servers (AKA, edge servers), e.g., PCC [31], BBR [23], QUIC [51], etc.

The optimal choice of configurations is contingent on
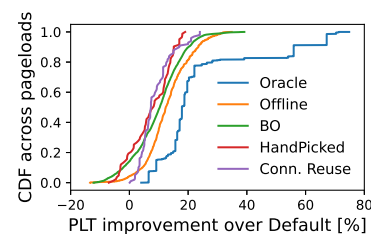
Figure 1: Comparison of various tuning techniques.

the network infrastructure [3, 36, 77, 98, 120, 135, 142], website complexity [20, 21, 95, 134, 136], and end-user devices [1, 94, 108]. Furthermore, innovations along any one of these dimensions will lead to changes to default parameters and new protocols. Although different regions and ISPs have radically different networking infrastructure and mobile devices [1, 93], a majority of CDNs continue to employ a "one-size-fits-all" [49] approach to configuring their edge servers, which results in sub-optimal performance [3, 36, 135] and high tail-latency in certain regions [142].

1.1 Configuration Tuning Status-Quo

Most attempts to tackle this growing diversity involve manually analyzing the performance of configuration options across different regions [49], devices [1], or websites [110, 135]. While several CDNs expose configuration knobs to their customers [40, 45], it is challenging to take the full advantage of the knobs due to the required manual efforts and the lack of automated learning techniques for effective tuning.

This paper focuses on tuning a broad set of configuration knobs across the transport (e.g., congestion control algorithm) and application layers (e.g., HTTP version) as highlighted in Table 1. Next, we illustrate the challenges and benefits of dynamically tuning network configurations.

Challenges in tuning stack: In Figure 1, we illustrate the difficulty of tuning configurations by comparing page load time (PLT) of popular websites, when configured using popular tuning techniques (setup explained in § 2.2). Specifically, *Bayesian Optimization* [104] (e.g., CherryPick [4]) a statistical

technique used for tuning systems configurations [4, 32, 75, 131], operator *hand-tuned* configurations (discussed in § 2), *TCP connection reuse* a traditional optimization (discussed in § 6.5), and a closed-loop *offline-learning* technique. We compare their performance against two baselines: *optimal* configuration discovered through an exhaustive brute-force search, and *default* configurations for Linux and Apache (Table 1).

Hand-tuned configurations are manually selected and are thus, coarse-grained. While they out-perform the *default* at median, they fail to provide optimal performance across varying network conditions and may even lead to performance degradation for some networks. TCP connection reuse only optimizes a subset of knobs (e.g., initial congestion window) and is unable to take full advantage of the diverse network stack knobs. Bayesian optimization aims to quickly discover "good" configuration. While fine-grained, this approach is relatively static and does not re-evaluate old choices, and is thus unable to adapt to network dynamics [78]. We observe the effects of this rigid behavior with wildly varying tail performance. Lastly, we explore an offline model which learns on traces from prior days and applies the learned model on connections for the next day. Offline modeling is fine-grained but with limited dynamicity: the trained model is unable to react to real-time issues. Unfortunately, due to the high dimensionality of the Internet's dynamics, these real-time issues are the norm, not the exception [67, 69, 78]. We observe in Figure 1 that offline performs closest to the optimal but still falls short because of its inability to react in real-time.

Our brief analysis of tuning approaches highlights the need for a dynamic, fine-grained approach to tuning configurations.

1.2 Configanator

In this paper, we eschew the notion of a homogeneous approach to tuning web server configurations and instead argue for a "curated" approach for configuring on a per-connection basis. In particular, we argue that edge servers should be configured to serve each of the incoming connections with the optimal protocols and configuration parameters, e.g., a web server may employ Cubic in favor of BBR when serving a low bottleneck buffer connection [111, 114]. To this end, we argue for a simple but robust server architecture that introduces flexibility into the network stack, enables reconfiguration, and systematically controls configuration heterogeneity. We also introduce a contextual multi-armed bandit based learning algorithm, an embodiment of domain-specific insights, which tunes configuration in a principled manner to find optimal configurations in minimal time. Taken together the design and the learning algorithm, our system, *Configanator*, enables a CDN to systematically explore heterogeneity in a dynamic and fine-grained manner while improving end-user performance. The design of Configanator faces several practical challenges:

- Network dynamics: network may change every few minutes [67, 83, 146] and thus requires continuous learning.

Layer	Protocol Options	Default	Example parameter
Transport	congestion_control (CC)	Cubic	BBR, Cubic, Reno
	initial congestion window	10 MSS	Integer (1, 4, 30)
	slow_start_after_idle	1	boolean {true, false}
	low_latency	0	boolean {true, false}
	autocorking	1	boolean {true, false}
	initRTO	1s	decimal (0.3,1)s [59]
	pacing (fair-queue)	0	boolean {true, false}
	timestamps	1	boolean {true, false}
	wmem	{4096}B	{163840}B
Web App	HTTP Protocol	1.1	1.1, 2
	H2 push	On	On, Off
	H2 max header list size	16384B	Integer values
	H2 header table size	4096B	Integer values
	H2 max concurrent streams	100	Integer values
	H2 initial window size	65535B	Integer values $< 2^{31}$
	H2 max frame size	16384B	Integer values $< 2^{24}$

Table 1: Web stack configuration parameters.

- Non-Gaussian noise: CDNs focus on improving tail latency [27, 53, 145] which is often caused by non-Gaussian processes (e.g., last-mile contention [125], mobile device limitations) and are difficult to model.

- High-Dimensionality: Content personalization, diverse devices [94, 134], and last-mile connections [125] introduce high dimensionality that limits the efficacy of offline closed-loop approaches [67, 68].

- High data cost: Generating data for learning requires testing configurations and may disrupt user's performance. Hence, the negative impact on users must be minimized.

- Limited flexibility: Linux kernel and modern web servers lack the flexibility to tune configurations on a per-connection basis, thus requires enhancing the traditional networking stack.

The key insight of Configanator is to simultaneously operate in two modes depending on the "quality" of the performance model. Essentially, Configanator intelligently selects samples that speed up model convergence, then at steady-state it transitions into a greedy-mode that stochastically samples points to iteratively improve performance. Configanator further clusters similar connections together and samples across clusters to amortize the cost of exploration.

Configanator uses a contextual multi-armed bandit [133] designed explicitly to continuously converge to an optimal (or near-optimal) configuration within a minimal number of exploration steps. Our ensemble fuses the stateful exploration of Gaussian-bandit with the non-determinism of Epsilon-bandit, enabling informed exploration of the configuration space while randomly re-sampling old configurations. The re-evaluation of data samples enables Configanator to directly tackle non-Gaussian noise within the domain. The data collected by the ensemble is encoded in a decision tree – which enables quick and easy classification but is also amenable to automatic generation of rules for a CDN's web server.

To demonstrate the benefits, we conducted large-scale simulations and live deployments. We used datasets from a *GlobalCDN* and public datasets from CAIDA [22], MAWI [8], Pantheon [142] and FCC [41]. Our simulation results show that Configanator provides 32-67% (up to 1500ms) improvement in the PLT at tail (p95) across the different traces. Given the

Layer	Option	Top configs. in N.A. (cross-CDN)	% of CDNs configuring differently across regions	Example of observed cross-regional difference
Web App	HTTP version	H1.1(44.3%), H2(55.7%)	4.7%	N.A. H2 -> Asia H1.1
	Max header list size	16384 (100%)	0%	None
	Header table size	4096 (100%)	0%	None
	Max concurrent streams	100 (44%), 128 (56%)	1%	N.A. 100 -> EU 128
	Initial window size	65536 (71%), 65535 (15%), >1M (14%)	1.9%	N.A. 1048576B -> Asia 65535B
	Max frame size	16,777,215 (81%), 16384 (19%)	0%	None
Transport	ICW	{10 (62%), 4(20.5%), 24(5.3%)} MSS	6.9%	N.A. 24 MSS -> Asia 10 MSS
	initRTO	{0.3(9.2%), 1(82.6%), 3(8.2%)} sec	2.3%	N.A. 3s -> EU 1s
	RWIN	{29200(57.4%), 14600(8.2%), 42780(6.8%)} bytes	3.6%	N.A. 29200B -> Asia 12960B

Table 2: Heterogeneity in configs. across 5 regions

recent arms race by CDNs to improve web performance, we believe that Configanator's modest improvements will result in significant revenue savings [14, 19, 30, 103]. Please refer to the project website [1] for the related resources.

2 Empirical Study

Next, we analyze CDNs to determine the current extent of configuration tuning (§ 2.1) and quantify its implications (§ 2.2).

2.1 Fingerprinting web configurations

We aim to understand if modern CDNs employ homogeneous configurations, as suggested by anecdotal evidence, or heterogeneous configurations to tackle diversity in the Internet's ecosystem. To this end, we developed a tool to infer and fingerprint a web server's [49, 92] application/L7 and transport/L4 layers configuration parameters by actively probing the servers and inspecting the packet headers and their reaction to emulated network events (e.g., packet loss). Please refer to Appendix A for more details about the tool. Using the tool, we fingerprinted the configurations for the Alexa top 1k websites from five different regions (North America (N.A), South America, Asia, Europe, and Australia), and present the results in Table 2. We use N.A configurations as the reference point and compare the observed configurations along two axes:

Observation 1: Heterogeneity across CDNs: In Column 3 (cross-CDN), we observe that different CDNs use different configurations in N.A. While some of the heterogeneity can be attributed to differences in the default values for different OSes, we observe that CDNs do use non-default values, e.g., amazon.com uses an ICW of 24 MSS in N.A.

Observation 2: Homogeneity within a CDN: In Column 4 (cross-region), we observe that only a small number of CDNs tune their network stack to account for regional differences, i.e., use different configurations in N.A. than the other regions. The highest amount of tuning occurs at L4, with 6.9% of the CDNs tuning the ICW differently in N.A. than in other regions, e.g., 24 MSS in N.A. but 10 MSS in Asia for amazon.de.

Takeaway: Taken together, these observations indicate that while individual CDNs perform modest tuning, most do not tune finely enough to account for regional diversity. In fact, only a small set of CDNs configure differently across regions.

2.2 Implications of Configuration Tuning

Next, we quantify the benefits of dynamically tuning a web server's networking stack by conducting a large scale study in our local testbed. We emulate a wide range of representative networks (extracted from real-world traces [8, 22, 41, 142]) and perform an exhaustive, brute-force search of configuration space (detailed description of the traces is provided in § 6.1). Table 1 lists the set of configurations, with default settings for TCP and HTTP taken from the Linux transport stack (kernel 4.20) and Apache (v2.4.18), respectively. In each trial, the server iteratively selects a configuration from the possible configuration space, a representative network is emulated using NetEM [54], and the PLT of a randomly selected website from Alexa Top-100 (locally cloned on the server) is measured five times. The *optimal* configuration is defined as the one that results in the lowest PLT for a specific network and website.

Figure 1 explores the implications of using sub-optimal configurations, by comparing optimal and default configurations for pageloads across diverse networks and websites. We observe that there is ∼18% PLT improvement at the median (over 70% at tail) when optimal configurations are used over the default. While the number may appear small, they can result in tremendous revenue improvements [14, 30], and more in the developing regions where CSPs are investing heavily to improve network [37, 80]. We observe the highest reconfiguration benefits for low bandwidth, high RTT/loss regions, representative of developing region networks.

Next, we analyze congestion control measurements across different regions from Pantheon [142]. We observe that emerging protocols, e.g., BBR, PCC, or Remy, which use probing or ML to improve performance, do not provide uniformly superior performance. In particular, we observed that in many situations BBR is suboptimal, performing 3X to 10X worse than the optimal congestion control. Moreover, no congestion control is optimal for more than 25% of the networks tested, and the median congestion control is optimal for only 6% of the networks.

3 Configanator's Algorithm

Tuning network configurations to maximize the web performance for diverse networks and end-users presents a complex learning problem. Next, we formulate the problem and present a domain-specific ensemble to address the challenges.

Problem Formulation: Given a set of networking configurations ($C=\{c_1, c_2...c_n\}$), network conditions (N

$= \{n_1, n_2...n_n\}$), devices ($D = \{d_1, d_2...d_n\}$), websites ($W = \{w_1, w_2...w_n\}$) and a function, $f()$, that maps a website, network condition, device, and configuration to a metric of web page performance (e.g., PLT or SpeedIndex). Note that, $f(c_i, n_i, d_i, w_i)$ returns the web page performance metric value for applying configuration c_i to a user device d_i loading website w_i in network n_i. In this paper, we use PLT as the metric for web page performance and can be easily replaced with other metrics. Our goal is to solve Eq. 1 and find a configuration ($c*$) that minimizes $f()$ for a given combination of n_i, d_i and w_i.

$$\underset{c*}{\operatorname{argmin}} f(c*, n_i, d_i, w_i) = \{f(c_i, n_i, d_i, w_i) | \forall c_i \in C\} \quad (1)$$

Solving the black-box function f() requires exploring sample space. Two possible exploration algorithms are:
- *Brute force* [2] which tests each possible configuration one by one until the entire space is explored.

- *Bayesian optimization* (BO) [16, 104] is a principled global optimization strategy that uses a prior probability function to capture the relationship between the objective function (Eq 1) and the observed data samples. BO models f(c,n,d,w) as a Gaussian process (GP) [16]. GP is a distribution of candidate objective functions and is used to select the next promising point (c*) which is then evaluated on a connection. GP then updates its posterior belief by adding the new observation f(c*,n,d,w) to the set of seen observations. With every new observation, the space of possible candidate functions gets smaller and the prior gets consolidated with the new evidence.

Challenges: Both approaches are sub-optimal for our use-case due to several reasons: (1) non-stationary network conditions [10, 67, 69, 146] (network conditions change every few minutes), (2) BO assumes that data is noise-free or only has Gaussian noise [118], and non-Gaussian noise (tail latency can not be modeled by a Gaussian process [78]) disrupts the estimation of next candidate sample and is observed to impact BO's hyper-parameters (e.g., threshold on expected improvement for next sample to stop the exploration), (3) costly data collection (collecting data requires testing on end-users which can impacts PLT and revenue), (4) data scarcity (testing on individual users requires each user to generate a tremendous number of connections but a user may only visit the site a few times).

Intuition: The intuition behind Configanator's algorithm is to decompose the model building into two phases: (i) an initial phase during which the search should be directed to speed up the process and build a good (not perfect) model, and (ii) a steady-state during which the search should be more stochastic to iteratively improve the model and tackle non-Gaussian noise. Building on these insights, Configanator leverages a combination of clustering, an ensemble of bandit-techniques, and ML to address the aforementioned challenges. Specifically, clustering is used to group connections based on their network and device similarity (called *Network Class*) and aggregate observations across similar connections to address data scarcity. The use of a contextual multi-armed bandit [133] enables Configanator to explore configurations and continuously collect data samples to learn and tackle dynamic client-side conditions in a balanced and online manner. To generalize observations across the connections, a Decision Tree is trained for efficient inference.

3.1 Domain-Specific Multi-Armed Bandit

Configanator's learning algorithm consists of a contextual multi-armed bandit [76, 84, 133] with three arms:

- **Exploration Arm-1 (Gaussian process [104, 112]):** The Gaussian process (GP) bandit [4, 73] uses an acquisition function to perform a directed search to quickly discover a "good" (might not be optimal) solution when no information exists for a Network Class (NC). There are multiple acquisition functions available [16] and we use *Expected Improvement (EI)* [112] because of its well-documented success [4, 32, 47]. This search process includes two terminating conditions: a threshold on EI and minimum of number of data points to explore. For non-continuous configurations (e.g., HTTP version), we encode them into a number to discretize the space[2]. To account for performance differences between websites and NCs, the GP-arm is composed of a collection of GP models, one for each unique website and NC combination (Appendix D).

- **Exploration Arm-2 (Epsilon-bandit [128]):** The Epsilon-bandit randomly re-samples the data points to overcome issues endemic with the Gaussian process (and Bayesian Optimization in general), e.g., non-stationarity of mean performance. The network operator bounds the random exploration by defining a parameter, ε, that controls the trade-off between speed of exploration and the impact on end-user QoE. A high ε improves exploration but results in a negative impact on clients' QoE due to constantly changing configurations.

- **Exploitation Arm (Decision Trees [107]):** The exploitation arm uses ML-powered prediction to model the data collected through the exploration arms. We evaluated several techniques including Support Vector Machines, Decision Trees (D-Trees), and Random Forests. We found that the D-Tree hits the sweet spot, providing comparable accuracy to the other models while being efficient enough to build and update at scale. The D-Tree encompasses all websites and NCs to learn across websites and networks. Leveraging the config-performance curves collected by underlying exploitation arms, a single D-Tree model is trained for the "good" configuration found so far for each website/NC pair, and the D-Tree maps {website, device, network/AS characteristics} to their optimal configuration.

Context-based arm switching: Configanator constantly switches between the arms based on the NC's "context" which is defined as the quality of the GP-model for the website/NC. It operates in two modes: (i) *Bootstrap*, when no information exists for a website or NC, the context is empty and the GP-arm is used to explore the configuration space in a principled manner until the acquisition function (EI) indicates

---

[2]GPyOpt [101] supports mixed (continuous/discrete) domain space [102].
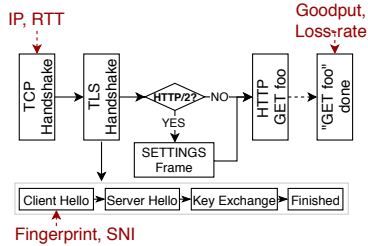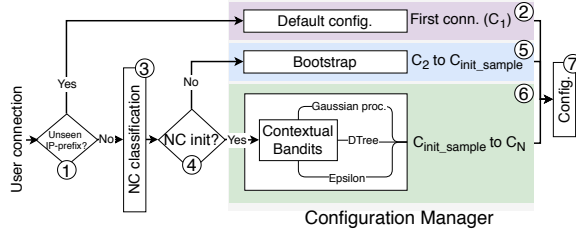
Figure 2: Connection features.
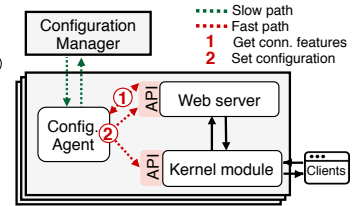

Figure 3: Learning framework workflow.

Figure 4: System architecture.

that a good configuration is found, (ii) *steady-state*, when information from the GP-arm indicates "good" configurations, Configanator uses either the epsilon-bandit to further explore the configuration space, or the exploitation arm (i.e., D-Tree) to leverage best configurations. Note that, random exploration through epsilon-bandit continues after EI threshold is met.

3.2 Discovering Network Classes

Configanator extends on observations from prior studies [67, 89] and classifies homogeneous connections into *Network Classes (NC)* with the intuition that similar connection characteristics lead to identical optimal configurations.

Design Goals and NC Features: The ideal NC-clustering should (i) create a small number of clusters, each with a large number of connections to amortize the cost of explorations, and (ii) all members of a cluster should have near-identical profiles. The two goals inherently contradict: the greater the number of entities in an NCs, the higher the probability that the NC contains entities with diverging performance. The second goal is further complicated by the sensitivity of a configuration's performance (e.g., PLT) to a myriad of factors in the end-to-end connection. To this end, we use network characteristics (bandwidth, latency, loss rate), AS information (ASN, geo-location), and device type as the basis for measuring similarity.

Capturing NC Features: To enable Configanator to effectively tune both the transport and HTTP layers, we must identify all features during the TCP handshake before the HTTP version is negotiated through ALPN [60]. If we identify features after HTTP negotiations, then tuning the HTTP layer would require renegotiation and hence incurs latency penalty. In Figure 2, we highlight the features collected during specific phases of the connection: (1) During the TCP handshake, we capture RTT, IP-prefix, and ASN/geo-location[3]. (2) During the TLS handshake, we apply TLS fingerprinting techniques [5, 18, 70, 127] on the TLS *Client Hello* to perform device identification and capture device features (accuracy evaluated in Appendix B). Note that, most operators already employ TLS fingerprinting for security purposes [6, 61, 126] and is also supported by major web servers [29]. We use the *Server Name Indication (SNI)* in the *Client Hello* to determine the website hostname which is one of the input features for the

learning framework. (3) For goodput and loss rates, features that cannot be captured during handshake, we build and use a historical archive of these network characteristics.

Network Classification: Clustering can be done using conventional techniques, e.g., K-means, hierarchical, or domain-specific techniques [17, 38, 105], e.g., Hobbit [74] or, CFA [67], or using CDN state of the art [26, 87, 119, 139, 140], e.g., latency-based groups [26, 119, 139]. Although Configanator can incorporate any of the aforementioned techniques, our prototype uses "K-means" clustering because of its simplicity. Configanator empirically selects the smallest K (i.e., the number of classes) that bounds the spread of performance within each NC by a predefined limit[4] (evaluated in § 6.2 and Appendix D).

3.3 Configanator Workflow

Figure 3 presents the end-to-end workflow. Default configuration is initially used for a newly-seen IP-prefix (①, ②) due to the lack of information about its goodput and loss-rates. For any subsequent connection from the IP-prefix, the recorded, as well as the actively collected features, are used for NC classification (③). If the network, AS and device characteristics do not fit into an existing NC, a new NC is created (④) and the next *init_samples* connections for the respective NC are used for bootstrapping (⑤) its empty context. When the respective NC is bootstrapped, the multi-armed bandit uses the actively and passively collected features, as well as the requested website, for determining the context and alternates between the arms (⑥). The connections are correspondingly tuned (⑦) and the resulting performance metrics are fed back into the models to help refine their classifications and improve accuracy. Due to the computationally intensive nature, Configanator builds/updates NC clusters in the background and uses the already-built clusters for real-time classification.

4 Architecture

Our re-architected web server consists of four components (Figure 4): The *HTTP server application* [39, 97, 121, 132] operates as it does today: serves content and collects performance metrics for each connection. The *Configuration Manager* runs the learning algorithm on the telemetry collected from the web servers. The *Configanator-API* abstracts vendor-specific

---

[3]Captured using end-user's IP and publicly available data (RouteViews for AS [17], MaxMind for geo-location [62])

[4]Controlled by *NCSpread* knob in simulator (Table 4 in Appendix).

configuration details and provides a uniform interface for configuring web server's network stack parameters. A *Configuration Agent* runs on each web server and uses the information received from the Configuration Manager to configure the connections through the Configanator-API.

Adopting this architecture in an incrementally deployable manner is practically challenging. The configuration parameters are exposed in an ad-hoc manner, e.g., tuning transport configuration requires *IOCTL* and *setsockopt*, while tuning HTTP requires changes to application code and enhancements to the ALPN protocol. Additionally, most CDNs use well-established code bases and exposing the configuration interfaces required by Configanator should incrementally build on the existing code.

4.1 Configanator-API

The *Configanator-API* presents a uniform interface over the web server's serving stack thus abstracting away OS and web server specific details. This simplified interface enables the *Configuration Agent* to easily tune the network stack, without having to understand vendor-specific details or implications.

Transport tuning: Unfortunately, the traditional kernels only expose and provide flexible reconfiguration for a subset of TCP's parameters. In particular, some parameters (e.g., ICW) can be configured on the connection level, while others can only be configured on a global scale (e.g., tcp_low_latency). Using Configanator at a coarser granularity, either limits the type of supported connections on a machine or limits the configuration space. There are several options to address this issue ranging from user-space TCP/IP stacks [35, 65, 106], kernel modules, eBPF programs, to leveraging virtualization. We opt for a kernel module-based design over virtualization approaches because hosting a single configuration per VM introduces significant overheads.

HTTP tuning: HTTP version and H2 settings are determined through Application Layer Protocol Negotiation (ALPN) [60] in TLS handshake and H2 SETTINGS [12] frame, respectively. Given the requirement for per-connection tuning, we augment the ALPN and the H2 SETTINGS frame code to enable fine-grained control over these configurations. In particular, Configanator configures these settings by restricting the options presented in the server advertisement to the configuration setting being tuned, e.g., to set the HTTP protocol to H2, we limit the "ALPN next protocol" field in TLS *Server_Hello* to just H2. Similarly, we restrict the options in the SETTINGS frame to configure HTTP/2 settings.

Tuning Workflow: Configanator-API tunes both the TCP and HTTP version during the TLS handshake: after receiving the *Client Hello* from the end-user and prior to sending the *Server Hello*. This is the perfect location to tune because (1) the complete feature set required to determine a connection's NC and configuration can be captured at this point, and (2) the server is yet to finalize the HTTP protocol, which the ALPN selects in *Server Hello*, thus enabling us to configure

the HTTP version. We note that at this phase of the connection, the TCP state machine is in its infancy because the sender has not sent any data, and thus virtually no significant state is lost when we change the congestion control algorithm or settings.

4.2 Configuration Agent

The *Configuration Agent* is the glue logic between the *Configuration Manager* and *Configanator-API* — it collects the connection features, uses rules provided by the Configuration Manager to make configuration decisions, and configures them using the *Configanator-API*. We select a proactive approach, where the *Configuration Manager* constantly pushes NC and configuration mappings to the *Configuration Agent* which caches them locally. Further for an unseen IP-prefix, *Configuration Agent* uses the default configuration, until the *Configuration Manager* finds a better mapping.

4.3 Configuration Manager

The manager runs in a centralized location, e.g., a data center or locally in a Point of Presence (PoP), with the implications later explored in Appendix F.9. It is charged with running the learning algorithms (§ 3), network classification models (§ 3.2), and disseminating the configuration maps to the *Configuration Agents*' cache. The *Configuration Manager* disseminates and collects data from the *Configuration Agents* using distributed asynchronous communication. For the NC and configuration maps, *Configuration Manager* broadcasts to all *Configuration Agents*, whereas for reporting performance data and for making one-off-request for configuration maps, the *Configuration Agents* use unicast.

5 Prototype

The implementation highlights of the prototype are as follows: **Configanator-API:** partly resides within the kernel (as a module) and partially resides in user-space in the form of additions to the web server code (in our case Apache). The components within the kernel allow us to tune the transport, while the user-space allows us to tune the HTTP layer.

The kernel module reuses functions provided by kernel's congestion controls through the *tcp_congestion_ops* interface and tunes fields in appropriate structs (e.g., *inet_connection_sock*). For tuning globally-defined knobs at a per-connection level (e.g., *tcp_low_latency*), we leveraged kernel patches [34, 55] to define and reference them from *tcp_sock* struct. The user-space component within Apache code tunes HTTP version in Apache and its design is generalizable to other servers that use OpenSSL. OpenSSL library is used by most web server implementations and allows web servers to register a *SSL_CTX_set_alpn_select_cb* [44] callback to modify ALPN decisions. To tune HTTP version, we register a call back which looks up the HTTP version to use for a connection and restricts the ALPN options advertised to the one specified by the Configuration Agent. For H2 settings, we modify the Apache H2

module to dynamically select the configurations while sending the SETTINGS frame. The user-space agent also generates the TLS fingerprint for device identification. We use JA3 [70] for TLS fingerprinting. In both our testbed experiments and in the live deployments, we use the ALPN-centric approach which modifies protocol options presented in the advertisements.

Configuration Agent: is user-space code and is implemented in 492 LoC of C++ code. The agent updates TCP and HTTP settings via the Configanator-API. This component also parses Apache's logs for measuring network characteristics. For measuring the PLT, the web server injects a simple JavaScript into the webpage to measure the navigation timings.

Configuration Manager: is developed in 1435 LoC (Python). It uses SciLearn [116] for D-Tree and GPyOpt [101] for the Gaussian Process. For communication with the Configuration Agents, we use ZeroMQ [144]. For D-Tree, we use SciLearn's CART algorithm with the following configuration: (i) entropy for the information gain, (ii) set the minimum number of leaf nodes to 80, (iii) set the minimum number of samples needed for the split to 2, and (iv) do not limit the depth of tree. For Gaussian process, we use init_sample=4, min_sample_tested=7 and EI=8% thresholds. We tested a range of these hyper-parameters settings [5] and selected the ones resulting in the highest accuracy. Following [4], we tested EI threshold in 3-15% range and selected 8% for its best trade-off between accuracy and search cost. For controlling the "K" for NCs, we use a *NCSpread* threshold of 5% (Appendix D).

6 Evaluation

We evaluated Configanator through a large-scale, trace-driven simulator using real-world traces, and live-deployments (§ 7). The simulation enables us to understand the system behavior under dynamic conditions, as well as analyze the implications of individual design choices.

6.1 Large Scale Trace Driven Simulations

Datasets: To simulate client activity, we use data from five sources: (i) *GlobalCDN* comprises 8.2M requests sampled from web and video services from 3 GlobalCDN PoPs (two in N. America, one in Europe) for a duration of 6 hours. Each request is a client fetching an object (e.g., web object, video chunk, etc.) and contains user information (IP prefix, ASN, etc.), observed server metrics (goodput, RTTs, loss rates etc.), CDN logs (e.g., user to edge PoP mapping [26, 119]) and performance metrics (time-to-last-byte). (ii) *CAIDA* [22], packet traces from the Equinix data-center in Chicago (in 2016). (iii) *MAWI* [8], packet traces from the WIDE backbone in Japan (in 2017). (iv) *FCC* [41], a U.S. nation-wide home broadband dataset. (v) *Pantheon* [141, 142], a data set of client sessions across different regions.

Generating client sessions: We use our traces to characterize the network conditions of real-world users. CAIDA and MAWI traces are captured at a vantage point between the client and server and we measure the goodput, RTT and loss rate by sequence-matching the data packets with their ACKs [6]. GlobalCDN[7], FCC and Pantheon datasets include the end-to-end network characteristics between a client and server. We model a client session as a time series of bandwidth (goodput), latency and loss rate measured between a pair of end-points.

Configuration Rewards: To avoid the pitfalls of trace-driven simulations [11], we decouple the modeling of configuration rewards (i.e., PLT calculation) from the process of generating client sessions. Our testbed comprises a cluster of 16 Linux servers (kernel 4.20), divided evenly to act as server (Apache) and clients (Chromium [50]). Our control over the machines and network enable us to set arbitrary server-side configurations (from Table 1) and emulate the bottleneck link to match the measured goodput, latency and loss rates from the datasets (using NetEm, TC [54]), with buffer-size set to Bandwidth Delay Product (BDP). To isolate the impact of network and configuration on PLT, caching (server or browser) is disabled and each server serves a single client (no resource contention). Using this testbed, we exhaustively measure the PLT for all combinations of configurations (Table 1). For each {network condition, configuration} pair, each website is loaded multiple times with the browser[8]. The final results are stored in a large tensor that maps {network condition (goodput, RTT, loss-rate), configuration, website} to PLT – called the *PLT-Tensor* comprising data from the pageloads in the testbed.

Simulator (Virtual Browser): Leveraging the client sessions and the PLT-Tensor, the simulator simulates the client's browsing behavior and interaction with Configanator as follows (visualized in Appendix F.1): (i) website[9], user information (e.g., IP) and session characteristics are taken as inputs, (ii) the learning framework determines the appropriate configuration for a connection, and (iii) pageload is simulated by using the PLT-Tensor to determine PLT for the client given the selected configuration. The simulator feeds the PLT back to the learning framework to complete the feedback loop.

PLT is sensitive to a myriad of features, ranging from dynamic network conditions at different time-of-the-day [67], user devices, to inherent variability. The session time-series captures the network dynamics and the testbed isolates the impact of network conditions on configurations. Table 4 lists the set of knobs we leveraged to test various realistic design choices, e.g., *NCSpread* to test various "K" sizes, *PerfMemory* to test the impact of PLT variability, etc. To account for the other factors like user devices, we conducted a scaled-down

---

[5] (e.g., entropy vs Gini impurity for information gain, number of leaf nodes ranging from 50 to 500, ID3, C4.5, and CART for D-Tree)

[6] Over a 5-second window (tunable through *ChunkSize* parameter in the simulator), e.g., data ACKed in 5s is used to measure goodput between vantage point/user. Further, we ignore duplicate ACKs while measuring RTTs.

[7] Measurements are on a per-request granularity. For multiple readings within the 5s window, we aggregate and use the median value.

[8] We repeated each measurement 5 times, similar to [135].

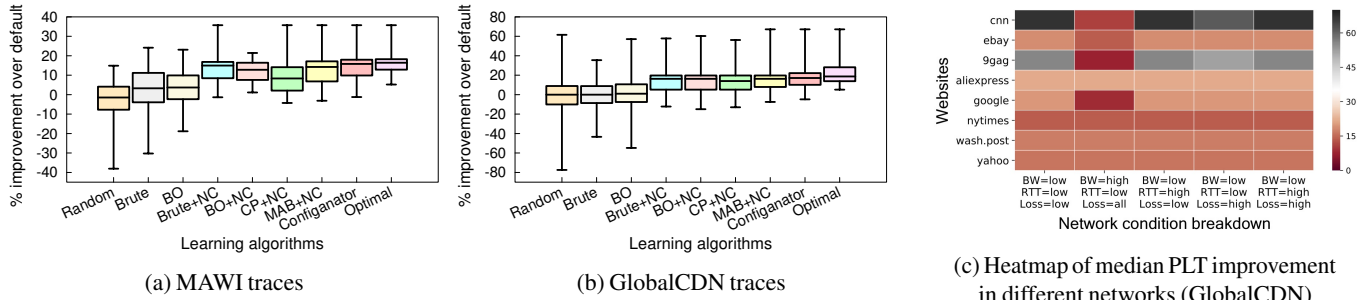[9] We iteratively load every website from our corpus for a given session.

Figure 5: Benefits of Configanator (box whiskers show 5th and 95th percentiles).

experiment on a CDN and tested different configurations for the real-world, diverse user devices (results in § 7.1).

Alternate algorithms: We evaluate against 8 algorithms:
(i, ii) Brute-force (Brute, Brute+NC): explores individual configurations, in an online manner, until all are explored and uses the best one (i.e., lowest PLT) for subsequent connections. *Brute* learns at the granularity of individual clients (i.e., unique IP) while *Brute+NC* clusters clients into Network Class (NC), and thus learning is spread across each NC.
(iii, iv, v) Bayesian Optimization (BO, BO+NC, CherryPick+NC): Bayesian Optimization is used to explore the configuration space and the best-explored option is exploited once BO-specific thresholds are met (§ 5). *BO* learns per client and *BO+NC* learns on a user group (NC) granularity. *CherryPick+NC* is similar to *BO+NC* but with hyper-parameters specified in [4].
(vi) Multi-armed Bandit (MAB+NC): uses traditional MAB with a weighted epsilon-greedy agent [133]. Each arm of the bandit is a different configuration, tested on NC granularity.
(vii) Random: Randomly selects a configuration in each trial.
(viii) Optimal: An oracle suggests the optimal parameters for a session by offline brute-force , i.e., PLT is calculated for the entire configuration space for each session offline and the configuration with the lowest PLT is used. This process is repeated for every session and puts an upper bound on improvement.

6.2 Effectiveness of Configanator

Figures 5a, 5b present the improvement in PLTs over default configurations for the different algorithms. The box plots compile data across the website pageloads for the client sessions in the respective trace. Configanator outperforms all alternatives at median and tail, improving p95 PLTs by 67% for GlobalCDN (1500ms), 36% (1100ms) for MAWI, 32% (610ms) for FCC, 48% (640ms) for CAIDA and 57% for Pantheon (850ms). Unlike Default, while Brute and BO apply different configurations to users, they assume that the network remains static and are unable to adapt to fluctuations. Moreover, due to its inability to adjust to fluctuations, BO often explores over 90% of the space without achieving the target EI, behaving similarly to *Brute*. Brute+NC, BO+NC and CherryPick+NC improve over the prior by amortizing the costs of learning but fail to adjust to non-Gaussian variations.

Although *MAB+NC* is able to handle non-Gaussian noise, it explores/exploits on a per-NC basis and, due to the lack of a cross-NC exploitation arm (Configanator's DT), *MAB+NC* falls short in its ability to apply patterns learnt across NCs.

As Configanator continuously learns and tests new configurations in an online fashion, a 'bad' configuration may be tested during the exploration phase and may lead to performance degradation. This behavior contributes to the worse PLT than Default for the p5 pageload in Figures 5a, 5b. A breakdown of the performance degradation and its causes are presented in Appendix F.8.

Dissecting Performance Improvements: Next, in Figure 5c, we analyze performance breakdown for a subset of websites according to the networking conditions used in prior work [135]). We make two observations: (i) Improvements tend to be higher in low bandwidth, low to high RTT/loss networks (typical for developing regions) with a median value of 14-67% compared to 10-25% for high bandwidth. We postulate that this trend is an outcome of the higher focus on developed region networks (typically high bandwidth, low RTT/loss) for the default configuration selection [33]. We observe a similar trend across our traces: GlobalCDN, MAWI and Pantheon traces (p95 RTT in 100-180ms) tend to show higher improvements than FCC and CAIDA (US-based, ∼60ms p95 RTT). (ii) the websites with highest benefits tend to be content-rich, e.g., 9gag.com and cnn.com observe >45% and >60% improvement, respectively, for all low bandwidth networks.

6.3 Benefits of Learning Ensemble

Next, we analyze the convergence for the top-3 algorithms from § 6.2 to focus on the aspects of Configanator that lead to better performance. We further split Configanator into two versions to analyze the benefits of its bandits: "NoGP" lacks GP and guided exploration, while "NoDT" lacks the decision tree.

Figures 6 plots the median distance from optimal across all NC and websites, for the first 500 update iterations. The observations are: (i) As data is gathered, Configanator performs better than others because of its ability to blend the benefits of both GP and DT – essentially efficient exploration and effective exploitation (iterations 3-10). Brute+NC exhaustively explores the complete space before converging to a choice, while MAB+NC exploration lacks the guided nature of acquisition
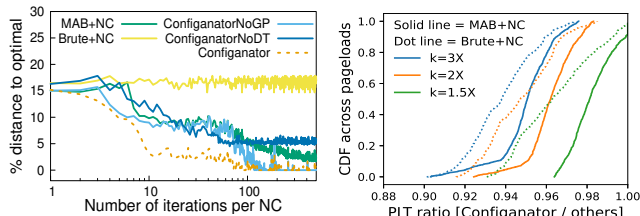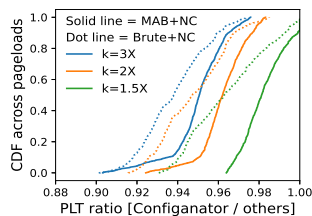
Figure 6: Cold-start convergence to optimal across NCs.

Figure 7: Impact of number of NCs (K) on performance.

function. (ii) Eventually, with sufficient data Configanator-NoGP is able to use the decision tree's predictive power to achieve near ideal performance (iterations 100+). Although MAB+NC gets within 2-3% of optimal for these iterations, it still needs more iterations to reach the optimal. (iii) While NoGP perform comparably for median at 200+ iterations, performance at the tail is **still** different (Appendix F.4).

6.4 Impact of Network Classes

Impact of Number of NCs: Next, we evaluate the impact of our clustering configuration (i.e., *NCSpread*) and analyze how the cluster size impacts performance. Intuitively, *NCSpread* bounds the performance variance within a cluster and has a direct impact on the number of clusters, or 'K'. Given a *NCSpread* value, the simulator performs a brute-force search to determine the smallest K that yields the threshold. We tested three scenarios with K inflated to {1.5, 2, 3} times the baseline value (Figure 5 experiments). The inversion from *NCSpread* to K and its implications on modeling accuracy are further discussed in Appendix D.

Figure 7 plots the ratio of Configanator and {MAB+NC, Brute+NC} PLT across the pageloads in GlobalCDN trace (<1 when Configanator outperforms). We observe the performance gap between Configanator and others increases with the K size. Although the large K results in a higher number of tighter NCs with lower performance spread within their constituents, it leads to an overall increase in exploration steps for MAB+NC and Brute+NC, as these algorithms explore the individual NCs independently. Further, the individual NC's best-found configuration is exploited for a narrower set of connections due to a lower number of connections in each cluster as compared to the case when K is small. On the other hand, the DT-arm in Configanator builds on the data collected for *all* NCs (§ 3.1). As soon as Configanator switches to DT-arm fairly early (Appendix F.3), it is able to exploit the best-found configuration for a wider audience, irrespective of the NC boundaries. The higher degree of exploration required by MAB+NC and Brute+NC makes their performance sub-optimal for the NCs with a smaller number of connections. Moreover, this can also lead to performance problems for tail connections, who are often in smaller NC due to their divergent network and device characteristics.

Impact of Size of NC (# of connections): Configanator aggregates network measurement across similar connections

and assumes homogeneity within an NC. Though an NC with a small number of users may lead to a smaller number of connections to learn from, it also favors the system as connections in the respective NC are strictly homogeneous. Next, we explore the impact of this bias on our results. We divide the NCs based on their unique number of IP prefixes and compare the PLTs observed for the individual prefixes with the NC's global PLT, i.e, median across all the prefixes in the NC. For two of such divisions, Table 3 presents the PLT comparison across the prefixes in NC groups. Compared to the <=5 group, i.e., NCs with a small number of distinct prefixes, where performance for most prefixes matches the global one; >=30 group shows more varying performance (e.g., lower than global PLT for the p25 prefix). However, we observe that the presence of larger NCs does not drastically impact Configanator as performance for most of the prefixes is still on par with the global one. For the tail prefixes that performs poorly as compared to the global PLT for the >=30 group, Configanator overfits the best-found configuration for the NC majority to the tail prefixes, and is observed to still outperform the *Default* (row 4 and 6 in Table 3).

6.5 TCP Connection Reuse (*ConnReuse*)

CDNs typically employ *ConnReuse*, allowing a new request to reuse older TCP connection. The key advantage of this feature is that the new request inherits matured congestion window (*cwnd*) and does not restart the connection from scratch, i.e. ICW. To analyze connection reuse, we analyzed the trace (GlobalCDN) to identify if and when requests reused existing connections and modified our setup to employ the reused connection's *cwnd* as the ICW for the page load[10].

Figure 8 plots Configanator improvement over *ConnReuse*. We observe that Configanator gains are reduced from 18% over *Default* (Figure 5b) to 14% at the median. The benefits at the tail are still substantial, with 56% p95 improvement. There are several reasons for this behavior: First, connection reuse only impacts the slow-start phase (e.g., ICW) and does not tune the CCA and HTTP, the top two critical knobs (Figure 13). Second, even with reuse, a connection is not always guaranteed to reuse the old *cwnd*, since other TCP settings like *slow_start_after_idle* may reset to default ICW — forcing a reused connection to again go through slow start phase. In fact, the old *cwnd* is reused with a probability of 0.27 in our trace, i.e., only a small subset of requests exploit the benefits of reuse. Third, unlike Configanator's exploitation of good configurations for similar connections, the scope of *ConnReuse* if limited to a single connection — a new connection from even the same user will go through the default slow start phase. Consequently, while connection reuse outperforms the *Default* by only 4.65% and 19.6% at median and tail respectively, a variant

---

[10]We infer *ConnReuse* if the first *cwnd* for a request is greater than connection's ICW. Our GlobalCDN trace directly captures these fields for each request. Note that, the reused connection may also inherit the MTU and SRTT values, but we limit our focus to the key component that limit data transfer, i.e., *cwnd*.

NC group	PLT ratio	Prefix distribution				
		p5	p25	p50	p75	95
<=5	Global / Configanator	0.92	1.0	1.0	1.0	1.08
	Default/Configanator	1.05	1.07	1.14	1.27	2.34
>=30	Global / Configanator	0.89	0.96	1.0	1.0	1.06
	Default/Configanator	1.04	1.09	1.17	1.21	2.67

Table 3: Impact of NC size on performance.

Figure 8: Impact vs TCP connection reuse.

Figure 9: Impact on fairness.

Figure 10: Only fair configurations.

of Configanator that only tunes *ICW* still performs *ConnReuse* with 8.7% and 23.4% improvements at median and tail.

These results suggest that *ConnReuse* alone is not the silver bullet, also portrayed by other ICW tuning system [42], and Configanator is expected to bring substantial improvements even when traditional optimizations are considered.

6.6 System Benchmarks

Next, we evaluate the latency, CPU and memory overheads. The experiments are performed in a testbed by emulating network conditions from our traces for 10 randomly selected websites. We repeat each test 1000 times.

Latency Overheads: For latency overheads, we focus on the modifications to ALPN to enable HTTP level tuning. We compare Configanator against a version that does not modify ALPN and tunes HTTP level by renegotiates which incurs at least 1-RTT overhead. Figure 11 plots the PLT for the two variants, normalized by *Default* (vanilla Apache). Given that Configanator simply edits the "ALPN next protocol" field in TLS *Server_Hello* without requiring any extra communication, we observe no latency overheads and a similar performance to the *Default*. For *Renegotiation*, we observe a slight PLT inflation ($\sim$ 3% at the median) which is due to the TLS renegotiation required to switch the HTTP version. We note that this approach still has a minor overhead (4% higher PLT at median) because a page load requires many RTTs and this overhead get amortized.

CPU and Memory Overheads To measure the CPU and memory overheads, we leveraged the Apache Benchmark tool to setup 100, 250 and 500 concurrent connections. We observe slight CPU overhead ($< 5\%$) as compared to *Default*. Although reconfiguring the connections do not require any additional memory, keeping the IP prefixes and their NC/configuration rules in the KV-store contributed to an increased memory usage.

6.7 Fairness Implications

Next, we explore the fairness implications. Within the testbed, we explore the situation where 30 concurrent flows share a representative bottleneck link, i.e., the access links for 3G, 4G, etc (number of flows from [115] Appendix F.10), under shallow buffers ({0.5 and 1} BDP). We use *Jain's Fairness Index* [63] to quantify fairness. We split the connections into two groups – one using Configanator and another using the default configuration (e.g., Cubic with 10MSS ICW). We then vary the percentage of connections in each group.
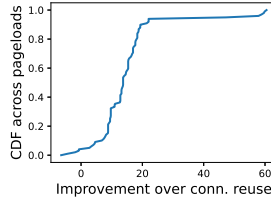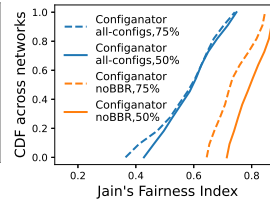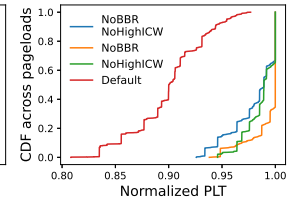
Quantifying Unfairness: Figure 9(a) present Jain's index when 75% and 50% of the flows are tuned. We observe that fairness decreases as the percentage of Configanator-tuned flows increases. Unsurprisingly, unfairness arises for two reasons: (1) when a flow is configured to use BBR [24,57,111,129,137], and (2) when a flow is configured to use high ICW values (even if BBR is not used) [52,72,88].

Configanator without unfair configurations: Next, we excluded the unfair configurations from the configuration space and tested 3 scenarios: (i) prevent BBR usage, (ii) prevent high ICW usage, (iii) prevent both. Figure 10 plots the ratio of PLT seen for vanilla Configanator (all configuration) to the variants, for GlobalCDN traces. We observe that NoBBR and NoHighICW perform similar to vanilla system for a significant fraction of the trace (63% and 35% respectively) and within 6% for worst case: this is because BBR is not always the optimal choice and application layer tuning (HTTP version) helps account for the lack of BBR or HighICW.

The results show that Configanator can provide an alternate war-chest to CDNs to improve web performance, even without using the unfair configurations.

6.8 Critical Knobs

We analyze the relative importance of reconfiguring different configuration parameters (Table 1). Our goal is to understand the minimal (or critical) parameters that must be tuned to significantly improve performance. In Figure 13, we plot the performance benefits of using distinct subset of configuration parameters, leveraging the brute-force exploration data from PLT-Tensor. We observe that the top 3 crucial parameters are HTTP version, congestion control algorithm (TCP-CC) and ICW. Moreover, when performing a layer to layer comparison, we observe that the Transport layer parameters combined (*Tran. layer*) have a higher impact on performance than the Application layer knobs combined(*App layer*). To explain this discrepancy, we analyze the different knobs in each layer and we observed that while certain transport knobs, e.g., Auto Corking, have little benefit in the median scenario, they are influential at the tails. Unlike the transport layer, in the Application layer most of the parameters (e.g., HTTP2 settings like header table size etc.) do not show significant benefit in median or tail conditions.
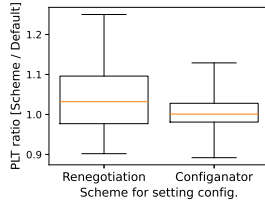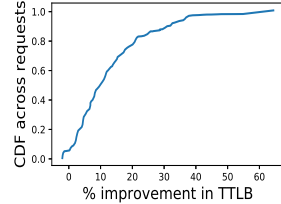
Figure 11: Latency overheads.
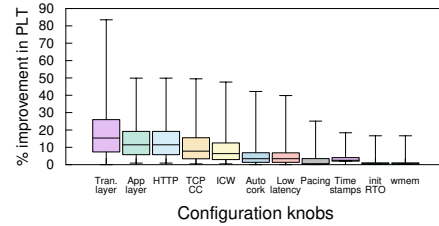

Figure 12: Reconfiguration benefits for CDN traffic

Figure 13: Critical configuration parameters

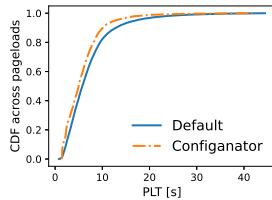| Pageload	PLT diff. (ms)	% imp.
p25	671	13.7
p50	767	14.6
p75	1219	21.5
p95	3797	26.3

Figure 14: Live deployment PLTs and improvements.

7 Live Deployment

In this section, we present the results for dynamically tuning the configurations at scale through a controlled experiment at GlobalCDN and a live prototype deployment on Google Cloud with 3161 end-users.

7.1 Validation at GlobalCDN

Next, we validate our approach when applied to data with more realistic and diverse client settings. We conducted measurements at GlobalCDN to collect data regarding performance of different configurations for the diverse networks and end-user devices. Specifically, we used the *default* configuration for 80% of the connections and explored random configurations for the rest to generate the data needed to emulate the contextual multi-armed bandit based exploration. Due to operational constraints, we analyzed a subset of configuration knobs: different congestion control algorithms and ICW. The experiments were conducted by randomly selecting 1% of the users from 3 of the CDN PoPs, for a duration of 6 hours. Note, these PoPs have the same workloads as the GlobalCDN trace described earlier.

We replayed the captured traces in our simulator, with two key distinctions: (i) the testbed-based PLT-Tensor was replaced by TTLB measurements collected from production users, since these TTLB measurements encompass the performance across real-world users, (ii) This experiment covers diverse user devices in-the-wild. Figure 12 presents the TTLB improvements for Configanator (versus default configuration) with upto 37% improvement at the tail (p95). Although this simulation covers a smaller configuration space, the improvements affirm the efficacy of tuning at scale, working with the diverse set of NC features (user device, network, geo-location, AS).

7.2 Google Cloud Deployment

We deployed Configanator on several Google Cloud servers, each with 8 CPU cores and 32 GB of RAM. We evenly divide

the servers into two groups: one half with the Configanator-enhanced servers, while the other half with traditional Apache server. We cloned a variety of real-world websites from Alexa top-100 and hosted them on servers without sharding. We hosted the Configuration Manager on a dedicated instance.

For clients, we used SpeedChecker [81, 82], a platform for global Internet measurements with vantage points deployed across the globe. We had 3161 clients in total, spread across 4 of the continents. The clients periodically conducted pageloads from both the Configanator and the traditional web servers at the same frequency, resulting in ~150K pageloads in 21 days. Further details about SpeedChecker are provided in Appendix F.1.

Figure 14 plots the raw PLTs observed for the two systems, with the accompanying table summarizing the PLT difference and improvements. Due to the online nature of the exploration and learning, we observe PLT degradation for a small subset of pageloads: 4.3% of the pageloads faced upto -13% degradation. For the rest, Configanator resulted in significant improvements, with upto 3.8s improved PLT at the tail (upto 767ms for the median). Dissecting the improvements across networks and websites, we observe a trend similar to Figure 5c: low bandwidth, high RTT/loss networks and content-rich websites get the most benefits. For the top configurations, we observe no clear winner: top 5 covered 3 CCs (BRR, Cubic and Vegas), both HTTP versions, and ICW randing from 16 to 40. We observe a stark difference in the ICW values used by clients in developed regions (Europe, N.America), with higher ICW (30-50 MSS), compared to developing regions (16-24 MSS).

Most of the clients (~75%) are from N.America/Europe and the rest are geographically distributed which results in unbalanced Network Classes (NCs), leading to a higher share of traffic for the probes in N.America/Europe. Interestingly, NCs with the most number of pageloads, although showing good improvements (11-13% at median), are not the ones where we observe the highest benefits, owing to their good bandwidth, low RTT connections. We observe that the less-dense NCs still outperform Default (by more than 8% at median), since Configanator's exploitation arm is able to generalize to a modest extent by using data collected across all NCs.

8 Discussion and Limitations

Security and Equilibrium: Potential implications of self-learning systems include adversarial attacks [123] or oscillations. We are working to formulate the interactions

between different instances of Configanator (i.e., deployments by different CDNs) as a game-theoretic problem to understand our system at equilibrium.

Management Overheads: Dynamically reconfiguring the CDN's protocol stack complicates performance diagnosis. We plan to investigate methods for reducing this complexity, e.g., minimizing the number of active configuration combinations. Further, different configurations may vary in their resource-consumption at the CDN edge and we plan to investigate the configuration associated resource-overheads in the future.

Data Bias: Configanator's data-driven workflow can be impacted by the inherent biases of trace-driven systems [11], e.g., choice of configuration can have an impact on the feedback loop's decision features. We leave a more comprehensive analysis of biasness to future work.

Testbed Limitations: Owing the lack of cellular connections and devices in the testbed, our simulator is unable to emulate different end-user devices and cellular last-miles. Although the dataset from GlobalCDN covers diverse last-mile connections and devices, we plan to explore systematic approaches to incorporate this diversity in the testbed.

Trace Limitations: While several of our traces capture end-to-end behavior (GlobalCDN, Pantheon, FCC), two of our traces do not. Specifically, CAIDA and MAWI traces are from core router and we recreate end-to-end behavior by matching data with ACKs: this recreation can introduce some imprecision into our latency, loss and BW calculations.

NC Size Bias: As demonstrated in the evaluation, connection homogeneity within an NC (due to small NC size) favors Configanator. This bias is prevalent in two of our traces, FCC and Pantheon (comprising synthetically generated flows). However, this does not hold for the realistic traces (e.g., GlobalCDN) which are mainly focused in the evaluation when discussing the size bias, and still shows improvements over the Default.

9   Related Work

Web Performance Many measurement studies [3, 33, 36, 48, 135] have explored the performance of different networking protocol settings and the impact of tuning on web performance. Our system builds on the observations from these studies: namely that different configurations are required for different network conditions and websites. Web improvement by cross-layer tuning was earlier motivated in [91] (Configanator's workshop paper) and the present paper builds a practical algorithm and system for tuning the configurations. It further evaluates idea in a wide range of realistic scenarios.

Self-Tuning Systems: Self-tuning systems have been explored within the context of transport protocols [31, 64, 77, 98–100, 113, 138], video [2, 68, 85, 124], databases [32, 56, 131], and cloud systems [4, 13, 78, 147]. While our work shares a similar ideology of exploiting heterogeneity, we differ in our methods for learning optimal configuration and in the domain specific solution for implementing

reconfiguration. While [68, 85] employ similar multi-armed bandits, our bandit generalizes across clusters and includes a Gaussian process to speed up learning. Additionally, while some model relatively static or offline workloads [2, 4, 32, 131], Configanator takes an online approach to tackle network and workload dynamics. Unlike [138] which rely on priori assumptions of the network, Configanator builds a performance model-based on live feedback which allows it to adapt to network dynamics. In contrast with [31, 64, 77, 79, 98–100, 113] which focus on tuning specific aspects of stack, Configanator tunes across a broader set of layers and parameters. Similarly, while these techniques use features from only network, Configanator also incorporates application features (e.g., website).

While Configanator focuses on control over server configurations, others [48, 109] require control over both the servers and the network switches to perform appropriate learning and tuning — applicable to data centers. Others [7, 90] move CCA outside data-path, enabling fast development and portability. Such innovative techniques simplify the design of Configanator by externalizing and simplifying tuning.

CrossLayer Optimizations. We differ from existing cross-layer optimizations [3, 9, 15, 25] which introduce APIs to enable the different layers to communicate and react accordingly the network events. Instead, we externalize the optimization logic and present an interface across the different layers to enable an external entity to configure the different layers which requires a learning algorithm agnostic of applications – a key contribution of Configanator.

10   Conclusion

In this paper, we argue that "one-size-fits-all" approach to configuring web server's network stack results in sub-par performance for end-users, especially those in emerging regions. Due to the ever-expanding nature of Internet, all end-users do not face similar network conditions. This argument stands in stark contrast to the traditional setup of today's web serving stacks where a single configuration is used for a divergent set of users.

This paper takes the first step towards realizing heterogeneity and fine-grained reconfiguration in a principled and systematic manner: our system, Configanator, introduces a principled framework for learning better configurations, than the default, for a connection by systematically exploring the performance of different configurations across a set of similar connections. We demonstrate the benefits of Configanator using both a live deployment and a large scale simulation.

11   Acknowledgments

References

[1] AHMAD, S., HAAMID, A. L., QAZI, Z. A., ZHOU, Z., BENSON, T., AND QAZI, I. A. A view from the other side: Understanding mobile phone characteristics in the developing world. In *Proceedings of the 2016 ACM on Internet Measurement Conference* (2016), ACM, pp. 319–325.

[2] AKHTAR, Z., NAM, Y. S., GOVINDAN, R., RAO, S., CHEN, J., KATZ-BASSETT, E., RIBEIRO, B., ZHAN, J., AND ZHANG, H. Oboe: auto-tuning video abr algorithms to network conditions. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (2018), ACM, pp. 44–58.

[3] AL-FARES, M., ELMELEEGY, K., REED, B., AND GASHINSKY, I. Overclocking the yahoo!: Cdn for faster web page loads. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference* (2011), ACM, pp. 569–584.

[4] ALIPOURFARD, O., LIU, H. H., CHEN, J., VENKATARAMAN, S., YU, M., AND ZHANG, M. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *NSDI* (2017), pp. 469–482.

[5] ALTHOUSE, J. Tls fingerprinting with ja3 and ja3s. https://sforce.co/3kUXKv8.

[6] ANDERSON, B. Tls fingerprinting in the real world. https://bit.ly/3l3Jnoe.

[7] ARASHLOO, M. T., GHOBADI, M., REXFORD, J., AND WALKER, D. Hotcocoa: Hardware congestion control abstractions. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks* (2017), ACM, pp. 108–114.

[8] ARCHIVE, M. W. G. T. Packet traces from wide backbone 12/1/17 to 12/7/17. http://mawi.wide.ad.jp/mawi/.

[9] BALAKRISHNAN, H., RAHUL, H. S., AND SESHAN, S. An integrated congestion management architecture for internet hosts. *ACM SIGCOMM Computer Communication Review 29*, 4 (1999), 175–187.

[10] BALAKRISHNAN, H., STEMM, M., SESHAN, S., AND KATZ, R. H. Analyzing stability in wide-area network performance. *ACM SIGMETRICS Performance Evaluation Review 25*, 1 (1997), 2–12.

[11] BARTULOVIC, M., JIANG, J., BALAKRISHNAN, S., SEKAR, V., AND SINOPOLI, B. Biases in data-driven networking, and what to do about them. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks* (2017), ACM, pp. 192–198.

[12] BELSHE, M., PEON, R., AND THOMSON, M. Hypertext transfer protocol version 2 (http/2). https://http2.github.io/http2-spec/.

[13] BILAL, M., AND CANINI, M. Towards automatic parameter tuning of stream processing systems. In *Proceedings of the 2017 Symposium on Cloud Computing* (New York, NY, USA, 2017), SoCC '17, ACM, pp. 189–200.

[14] BOJAN PAVIC, CHRIS ANSTEY, J. W. Why does speed matter? https://web.dev/why-speed-matters/.

[15] BRIDGES, P. G., WONG, G. T., HILTUNEN, M., SCHLICHTING, R. D., AND BARRICK, M. J. A configurable and extensible transport protocol. *IEEE/ACM Transactions on Networking 15*, 6 (2007), 1254–1265.

[16] BROCHU, E., CORA, V. M., AND DE FREITAS, N. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599* (2010).

[17] BROIDO, A., AND CLAFFY, K. Analysis of Route-Views BGP data: policy atoms. In *Network Resource Data Management Workshop* (Santa Barbara, CA, May 2001).

[18] BROTHERSTON, L. Fingerprintls. https://bit.ly/3BJfFuK.

[19] BRUTLAG, J. Speed matters for google web search, 2009.

[20] BUTKIEWICZ, M., MADHYASTHA, H. V., AND SEKAR, V. Understanding website complexity: measurements, metrics, and implications. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference* (2011), ACM, pp. 313–328.

[21] BUTKIEWICZ, M., WANG, D., WU, Z., MADHYASTHA, H. V., AND SEKAR, V. Klotski: Reprioritizing web content to improve user experience on mobile devices. In *NSDI* (2015), vol. 1, pp. 2–3.

[22] CAIDA. The caida ucsd anonymized internet traces 2016 dataset. https://bit.ly/3l1AVG6.

[23] CARDWELL, N., CHENG, Y., GUNN, C. S., YEGANEH, S. H., ET AL. Bbr: congestion-based congestion control. *Communications of the ACM 60*, 2 (2017), 58–66.

[24] CARDWELL, N., CHENG, Y., YEGANEH, S. H., SWETT, I., VASILIEV, V., JHA, P., SEUNG, Y., MATHIS, M., AND JACOBSON, V. Bbr v2 a model-based congestion control. https://bit.ly/3x4bQwx.

[25] CHEN, A., SRIRAMAN, A., VAIDYA, T., ZHANG, Y., HAEBERLEN, A., LOO, B. T., PHAN, L. T. X., SHERR, M., SHIELDS, C., AND ZHOU, W. Dispersing asymmetric ddos attacks with splitstack. In *HotNets* (2016), pp. 197–203.

[26] CHEN, F., SITARAMAN, R. K., AND TORRES, M. End-user mapping: Next generation request routing for content delivery. *ACM SIGCOMM Computer Communication Review 45*, 4 (2015), 167–181.

[27] CUI, Y., DAI, N., LAI, Z., LI, M., LI, Z., HU, Y., REN, K., AND CHEN, Y. Tailcutter: Wisely cutting tail latency in cloud cdns under cost constraints. *IEEE/ACM Transactions on Networking 27*, 4 (2019), 1612–1628.

[28] DEAN, J., AND BARROSO, L. A. The tail at scale. *Communications of the ACM 56* (2013), 74–80.

[29] DEV@TRAFFICSERVER.APACHE.ORG. Ja3 fingerprint plugin. https://bit.ly/3iTg70U.

[30] DIGITAL, D. Milliseconds make millions: A study on how improvements in mobile site speed positively affect a brand's bottom line. https://bit.ly/3rpm8WP.

[31] DONG, M., LI, Q., ZARCHY, D., GODFREY, P. B., AND SCHAPIRA, M. Pcc: Re-architecting congestion control for consistent high performance. In *NSDI* (2015), vol. 1, p. 2.

[32] DUAN, S., THUMMALA, V., AND BABU, S. Tuning database configuration parameters with ituned. *Proceedings of the VLDB Endowment 2*, 1 (2009), 1246–1257.

[33] DUKKIPATI, N., REFICE, T., CHENG, Y., CHU, J., HERBERT, T., AGARWAL, A., JAIN, A., AND SUTIN, N. An argument for increasing tcp's initial congestion window. *Computer Communication Review 40*, 3 (2010), 26–33.

[34] DUMAZET, E. tcp: provide syn headers for passive connections. https://lwn.net/Articles/645128/.

[35] DUNKELS, A., ET AL. The lwip tcp/ip stack. *lwIP–A LightWeight TCP/IP Stack* (2004).

[36] ERMAN, J., GOPALAKRISHNAN, V., JANA, R., AND RAMAKRISHNAN, K. K. Towards a spdyier mobile web? *IEEE/ACM Transactions on Networking 23*, 6 (2015), 2010–2023.

[37] FACEBOOK. Aquila (internet deployment by drone). https://bit.ly/2V92Adk.

[38] FACEBOOK. Network connection class. https://github.com/facebook/network-connection-class.

[39] FACEBOOK. Proxygen: Facebook's c++ http libraries. https://github.com/facebook/proxygen.

[40] FASTLY. Advanced tcp optimizations. https://bit.ly/3f2SstA.

[41] FCC. Measuring fixed broadband report - 2016. https://bit.ly/2TAxef8.

[42] FLORES, M., KHAKPOUR, A. R., AND BEDI, H. Riptide: Jump-starting back-office connections in cloud systems. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)* (2016), IEEE, pp. 78–87.

[43] FOUNDATION, L. *Open vSwitch*.

[44] FOUNDATION, O. S. Openssl alpn callback. https://bit.ly/3rG5rXh.

[45] FOWLER, D. Cdn tuning for ott - why doesn't it already do that? https://bit.ly/3i4z7Kr.

[46] GANJAM, A., SIDDIQUI, F., ZHAN, J., LIU, X., STOICA, I., JIANG, J., SEKAR, V., AND ZHANG, H. C3: Internet-scale control plane for video quality optimization. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)* (2015), pp. 131–144.

[47] GARDNER, J. R., KUSNER, M. J., XU, Z. E., WEINBERGER, K. Q., AND CUNNINGHAM, J. P. Bayesian optimization with inequality constraints. In *ICML* (2014), pp. 937–945.

[48] GHOBADI, M., YEGANEH, S. H., AND GANJALI, Y. Rethinking end-to-end congestion control in software-defined networks. In *Proceedings of the 11th ACM Workshop on Hot Topics in networks* (2012), ACM, pp. 61–66.

[49] GONG, S., NASEER, U., AND BENSON, T. Inspector gadget: A framework for inferring tcp congestion control algorithms and protocol configurations. In *Network Traffic Measurement and Analysis Conference (TMA)* (2020).

[50] GOOGLE. The chromium projects. https://www.chromium.org/.

[51] GOOGLE. Quic, a multiplexed stream transport over udp. https://www.chromium.org/quic.

[52] GRIECO, L. A., AND MASCOLO, S. Performance evaluation and comparison of westwood+, new reno, and vegas tcp congestion control. *ACM SIGCOMM Computer Communication Review 34*, 2 (2004), 25–38.

[53] HELT, J., FENG, G., SESHAN, S., AND SEKAR, V. Sandpaper: mitigating performance interference in cdn edge proxies. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing* (2019), pp. 30–46.

[54] HEMMINGER, S. Netem - network emulator. https://bit.ly/36ZXT8k.

[55] HERBERT, T. tcp: Socket option to set congestion window. https://bit.ly/3zDnH6r.

[56] HERODOTOU, H., LIM, H., LUO, G., BORISOV, N., DONG, L., CETIN, F. B., AND BABU, S. Starfish: A self-tuning system for big data analytics. In *Cidr* (2011), vol. 11, pp. 261–272.

[57] HOCK, M., BLESS, R., AND ZITTERBART, M. Experimental evaluation of bbr congestion control. In *Network Protocols (ICNP), 2017 IEEE 25th International Conference on* (2017), IEEE, pp. 1–10.

[58] HOFF, T. Latency is everywhere and it costs you sales - how to crush it. https://bit.ly/3x5u3Kb.

[59] IETF. Rfc 6298. https://tools.ietf.org/html/rfc6298.

[60] IETF. Rfc 7301 transport layer security (tls) application-layer protocol negotiation extension. https://tools.ietf.org/rfc/rfc7301.txt.

[61] INC., C. Malcolm measuring active listeners, connection observers, and legitimate monitors. https://malcolm.cloudflare.com/.

[62] INC., M. Geoip2 city database. https://www.maxmind.com/en/geoip2-city.

[63] JAIN, R., DURRESI, A., AND BABIC, G. Throughput fairness index: An explanation. In *ATM Forum contribution* (1999), vol. 99.

[64] JAY, N., ROTMAN, N. H., GODFREY, P., SCHAPIRA, M., AND TAMAR, A. Internet congestion control via deep reinforcement learning. *arXiv preprint arXiv:1810.03259* (2018).

[65] JEONG, E., WOO, S., JAMSHED, M. A., JEONG, H., IHM, S., HAN, D., AND PARK, K. mtcp: a highly scalable user-level tcp stack for multicore systems. In *NSDI* (2014), pp. 489–502.

[66] JIANG, J., DAS, R., ANANTHANARAYANAN, G., CHOU, P. A., PADMANABHAN, V., SEKAR, V., DOMINIQUE, E., GOLISZEWSKI, M., KUKOLECA, D., VAFIN, R., ET AL. Via: Improving internet telephony call quality using predictive relay selection. In *Proceedings of the 2016 ACM SIGCOMM Conference* (2016), ACM, pp. 286–299.

[67] JIANG, J., SEKAR, V., MILNER, H., SHEPHERD, D., STOICA, I., AND ZHANG, H. Cfa: A practical prediction system for video qoe optimization. In *NSDI* (2016), pp. 137–150.

[68] JIANG, J., SUN, S., SEKAR, V., AND ZHANG, H. Pytheas: Enabling data-driven quality of experience optimization using group-based exploration-exploitation. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (Boston, MA, 2017), USENIX Association, pp. 393–406.

[69] JIN, Y., RENGANATHAN, S., ANANTHANARAYANAN, G., JIANG, J., PADMANABHAN, V. N., SCHRODER, M., CALDER, M., AND KRISHNAMURTHY, A. Zooming in on wide-area latencies to a global cloud provider. In *Proceedings of the ACM Special Interest Group on Data Communication* (2019), ACM, pp. 104–116.

[70] JOHN B. ALTHOUSE, JEFF ATKINSON, J. A. Ja3 - a method for profiling ssl/tls clients. https://github.com/salesforce/ja3.

[71] KAYSER, B. What is the expected distribution of website response times?

[72] KOZU, T., AKIYAMA, Y., AND YAMAGUCHI, S. Improving rtt fairness on cubic tcp. In *2013 First International Symposium on Computing and Networking* (2013), IEEE, pp. 162–167.

[73] KRAUSE, A., AND ONG, C. S. Contextual gaussian process bandit optimization. In *Advances in neural information processing systems* (2011), pp. 2447–2455.

[74] LEE, Y., AND SPRING, N. Identifying and aggregating homogeneous ipv4 /24 blocks with hobbit. In *Proceedings of the 2016 Internet Measurement Conference* (New York, NY, USA, 2016), IMC '16, ACM, pp. 151–165.

[75] LETHAM, B., KARRER, B., OTTONI, G., AND BAKSHY, E. *Efficient tuning of online systems using Bayesian optimization.* https://bit.ly/3rBMHIm.

[76] LI, L., CHU, W., LANGFORD, J., AND SCHAPIRE, R. E. A contextual-bandit approach to personalized news article recommendation. In *Proceedings of the 19th international conference on World wide web* (2010), ACM, pp. 661–670.

[77] LI, W., ZHOU, F., CHOWDHURY, K. R., AND MELEIS, W. M. Qtcp: Adaptive congestion control with reinforcement learning. *IEEE Transactions on Network Science and Engineering* (2018).

[78] LI, Z. L., LIANG, M. C.-J., HE, W., ZHU, L., DAI, W., JIANG, J., AND SUN, G. Metis: Robustly tuning tail

latencies of cloud systems. In *ATC (USENIX Annual Technical Conference)* (July 2018), USENIX.

[79] LIU, H. H., VISWANATHAN, R., CALDER, M., AKELLA, A., MAHAJAN, R., PADHYE, J., AND ZHANG, M. Efficiently delivering online services over integrated infrastructure. In *NSDI* (2016), vol. 1, p. 1.

[80] LOON, P. Balloon powered internet. `https://x.company/loon/`.

[81] LTD., S. Speedchecker. `https://probeapi.speedchecker.com/`.

[82] LTD., S. Speedchecker - probe api documentation. `https://bit.ly/2TGwdCu`.

[83] LU, D., QIAO, Y., DINDA, P. A., AND BUSTAMANTE, F. E. Characterizing and predicting tcp throughput on the wide area network. In *Distributed Computing Systems, 2005. ICDCS 2005. Proceedings. 25th IEEE International Conference on* (2005), IEEE, pp. 414–424.

[84] LU, T., PÁL, D., AND PÁL, M. Contextual multi-armed bandits. In *Proceedings of the Thirteenth international conference on Artificial Intelligence and Statistics* (2010), pp. 485–492.

[85] MAO, H., NETRAVALI, R., AND ALIZADEH, M. Neural adaptive video streaming with pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017), ACM, pp. 197–210.

[86] MCKAY, M. D., BECKMAN, R. J., AND CONOVER, W. J. Comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics 21*, 2 (1979), 239–245.

[87] MCQUISTIN, S., UPPU, S. P., AND FLORES, M. Taming anycast in the wild internet. In *Proceedings of the Internet Measurement Conference* (2019), pp. 165–178.

[88] MO, J., LA, R. J., ANANTHARAM, V., AND WALRAND, J. Analysis and comparison of tcp reno and vegas. In *IEEE INFOCOM'99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No. 99CH36320)* (1999), vol. 3, IEEE, pp. 1556–1563.

[89] MUKERJEE, M. K., NAYLOR, D., JIANG, J., HAN, D., SESHAN, S., AND ZHANG, H. Practical, real-time centralized control for cdn-based live video delivery. *ACM SIGCOMM Computer Communication Review 45*, 4 (2015), 311–324.

[90] NARAYAN, A., CANGIALOSI, F., RAGHAVAN, D., GOYAL, P., NARAYANA, S., MITTAL, R., ALIZADEH, M., AND BALAKRISHNAN, H. Restructuring endpoint congestion control. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (2018), ACM, pp. 30–43.

[91] NASEER, U., AND BENSON, T. Configtron: Tackling network diversity with heterogeneous configurations. In *9th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 17)* (Santa Clara, CA, July 2017), USENIX Association.

[92] NASEER, U., AND BENSON, T. Inspectorgadget: Inferring network protocol configuration for web services. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)* (July 2018), pp. 1624–1629.

[93] NASEER, U., BENSON, T. A., AND NETRAVALI, R. Webmedic: Disentangling the memory-functionality tension for the next billion mobile web users. In *Proceedings of the 22nd International Workshop on Mobile Computing Systems and Applications* (2021), pp. 71–77.

[94] NEJATI, J., AND BALASUBRAMANIAN, A. An in-depth study of mobile browser performance. In *Proceedings of the 25th International Conference on World Wide Web* (2016), International World Wide Web Conferences Steering Committee, pp. 1305–1315.

[95] NETRAVALI, R., GOYAL, A., MICKENS, J., AND BALAKRISHNAN, H. Polaris: Faster page loads using fine-grained dependency tracking. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (2016), USENIX Association.

[96] NETRAVALI, R., SIVARAMAN, A., DAS, S., GOYAL, A., WINSTEIN, K., MICKENS, J., AND BALAKRISHNAN, H. Mahimahi: Accurate record-and-replay for http. In *USENIX Annual Technical Conference* (2015), pp. 417–429.

[97] NGINX. Nginx reverse proxy. `https://bit.ly/3yapgbH`.

[98] NIE, X., ZHAO, Y., CHEN, G., SUI, K., CHEN, Y., PEI, D., ZHANG, M., AND ZHANG, J. Tcp wise: One initial congestion window is not enough. In *Performance Computing and Communications Conference (IPCCC), 2017 IEEE 36th International* (2017), IEEE, pp. 1–8.

[99] NIE, X., ZHAO, Y., LI, Z., CHEN, G., SUI, K., ZHANG, J., YE, Z., AND PEI, D. Dynamic tcp initial windows and congestion control schemes through reinforcement learning. *IEEE Journal on Selected Areas in Communications 37*, 6 (2019), 1231–1247.

[100] NIE, X., ZHAO, Y., PEI, D., CHEN, G., SUI, K., AND ZHANG, J. Reducing web latency through dynamically setting tcp initial window with reinforcement learning.

[101] OF SHEFFIELD, M. L. G. U. Gpyopt. https://github.com/SheffieldML/GPyOpt.

[102] OF SHEFFIELD, M. L. G. U. Gpyopt.core.task.space module. https://bit.ly/3iVDuHf.

[103] OREILLY.COM. Bing and google agree: Slow pages lose users. https://bit.ly/374YGVI.

[104] PELIKAN, M., GOLDBERG, D. E., AND CANTÚ-PAZ, E. Boa: The bayesian optimization algorithm. In *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation-Volume 1* (1999), Morgan Kaufmann Publishers Inc., pp. 525–532.

[105] PI, Y., JAMIN, S., DANZIG, P., AND SHAHA, J. Ap-atoms: A high-accuracy data-driven client aggregation for global load balancing. *IEEE/ACM Transactions on Networking 26*, 6 (2018), 2748–2761.

[106] PICOTCP. picotcp. http://www.picotcp.com/.

[107] QUINLAN, J. R. Induction of decision trees. *Machine learning 1*, 1 (1986), 81–106.

[108] RUAMVIBOONSUK, V., NETRAVALI, R., ULUYOL, M., AND MADHYASTHA, H. V. Vroom: Accelerating the mobile web with server-aided dependency resolution. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017), ACM, pp. 390–403.

[109] RUFFY, F., PRZYSTUPA, M., AND BESCHASTNIKH, I. Iroko: A framework to prototype reinforcement learning for data center traffic control. *arXiv preprint arXiv:1812.09975* (2018).

[110] RÜTH, J., BORMANN, C., AND HOHLFELD, O. Large-scale scanning of tcp's initial window. In *Proceedings of the 2017 Internet Measurement Conference* (2017), ACM, pp. 304–310.

[111] RÜTH, J., KUNZE, I., AND HOHLFELD, O. An empirical view on content provider fairness. *arXiv preprint arXiv:1905.07152* (2019).

[112] RYZHOV, I. O. On the convergence rates of expected improvement methods. *Operations Research 64*, 6 (2016), 1515–1528.

[113] SCHAPIRA, M., AND WINSTEIN, K. Congestion-control throwdown. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks* (2017), ACM, pp. 122–128.

[114] SCHOLZ, D., JAEGER, B., SCHWAIGHOFER, L., RAUMER, D., GEYER, F., AND CARLE, G. Towards a deeper understanding of tcp bbr congestion control. In *2018 IFIP Networking Conference (IFIP Networking) and Workshops* (2018), IEEE, pp. 1–9.

[115] SCHULMAN, A., LEVIN, D., AND SPRING, N. CRAW-DAD dataset umd/sigcomm2008 (v. 2009-03-02). Downloaded from https://crawdad.org/umd/sigcomm2008/20090302/pcap, Mar. 2009. traceset: pcap.

[116] SCIKIT LEARN.ORG. *Decision Trees*.

[117] SERVER, A. T. Tcpinfo plugin. https://bit.ly/3x8CyEr.

[118] SHAHRIARI, B., SWERSKY, K., WANG, Z., ADAMS, R. P., AND DE FREITAS, N. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE 104*, 1 (2015), 148–175.

[119] SHUFF, P. Building a billion user load balancer. USENIX Association.

[120] SINGH, S., MADHYASTHA, H. V., KRISHNAMURTHY, S. V., AND GOVINDAN, R. Flexiweb: Network-aware compaction for accelerating mobile web transfers. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking* (2015), ACM, pp. 604–616.

[121] SOFTWARE, V. Varnish http cache. https://varnish-cache.org/.

[122] STEIN, M. Large sample properties of simulations using latin hypercube sampling. *Technometrics 29*, 2 (1987), 143–151.

[123] SUN, Y., EDMUNDSON, A., VANBEVER, L., LI, O., REXFORD, J., CHIANG, M., AND MITTAL, P. Raptor: Routing attacks on privacy in tor.

[124] SUN, Y., YIN, X., JIANG, J., SEKAR, V., LIN, F., WANG, N., LIU, T., AND SINOPOLI, B. Cs2p: Improving video bitrate selection and adaptation with data-driven throughput prediction. In *Proceedings of the 2016 ACM SIGCOMM Conference* (2016), ACM, pp. 272–285.

[125] SUNDARESAN, S., FEAMSTER, N., AND TEIXEIRA, R. Home network or access link? locating last-mile downstream throughput bottlenecks. In *International Conference on Passive and Active Network Measurement* (2016), Springer, pp. 111–123.

[126] TEAM, A. T. R. Bots tampering with tls to avoid detection. https://bit.ly/3f2cJjb.

[127] TLSFINGERPRINT.IO. Tls fingerprint. https://tlsfingerprint.io/.

[128] TRAN-THANH, L., CHAPMAN, A., DE COTE, E. M., ROGERS, A., AND JENNINGS, N. R. Epsilon–first policies for budget–limited multi-armed bandits. In *Twenty-Fourth AAAI Conference on Artificial Intelligence* (2010).

[129] TURKOVIC, B., KUIPERS, F. A., AND UHLIG, S. Interactions between congestion control algorithms. *network 3*, 17.

[130] URVOY-KELLER, G. On the stationarity of tcp bulk data transfers. In *International Workshop on Passive and Active Network Measurement* (2005), Springer, pp. 27–40.

[131] VAN AKEN, D., PAVLO, A., GORDON, G. J., AND ZHANG, B. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data* (2017), ACM, pp. 1009–1024.

[132] VDMS. Our software - cdn. https://bit.ly/3iOMNbT.

[133] VERMOREL, J., AND MOHRI, M. Multi-armed bandit algorithms and empirical evaluation. In *European conference on machine learning* (2005), Springer, pp. 437–448.

[134] WANG, X. S., BALASUBRAMANIAN, A., KRISHNAMURTHY, A., AND WETHERALL, D. Demystifying page load performance with wprof. In *NSDI* (2013), pp. 473–485.

[135] WANG, X. S., BALASUBRAMANIAN, A., KRISHNAMURTHY, A., AND WETHERALL, D. How speedy is spdy? In *NSDI* (2014), pp. 387–399.

[136] WANG, X. S., KRISHNAMURTHY, A., AND WETHERALL, D. Speeding up web page loads with shandian. In *NSDI* (2016), pp. 109–122.

[137] WARE, R., MUKERJEE, M. K., SESHAN, S., AND SHERRY, J. Beyond jain's fairness index: Setting the bar for the deployment of congestion control algorithms. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks* (2019), pp. 17–24.

[138] WINSTEIN, K., AND BALAKRISHNAN, H. Tcp ex machina: computer-generated congestion control. In *ACM SIGCOMM Computer Communication Review* (2013), vol. 43, ACM, pp. 123–134.

[139] WOHLFART, F., CHATZIS, N., DABANOGLU, C., CARLE, G., AND WILLINGER, W. Leveraging interconnections for performance: The serving infrastructure of a large cdn. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (2018), pp. 206–220.

[140] WONDRA, N. Magic transit: Network functions at cloudflare scale.

[141] YAN, F. Y., MA, J., HILL, G. D., RAGHAVAN, D., WAHBY, R. S., LEVIS, P., AND WINSTEIN, K. Pantheon datasets. https://pantheon.stanford.edu/measurements/node/.

[142] YAN, F. Y., MA, J., HILL, G. D., RAGHAVAN, D., WAHBY, R. S., LEVIS, P., AND WINSTEIN, K. Pantheon: the training ground for internet congestion-control research. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, 2018), USENIX Association, pp. 731–743.

[143] YANG, P., SHAO, J., LUO, W., XU, L., DEOGUN, J., AND LU, Y. Tcp congestion avoidance algorithm identification. *IEEE/ACM Transactions on Networking (TON) 22*, 4 (2014), 1311–1324.

[144] ZEROMQ. Zeromq. http://zeromq.org/.

[145] ZHANG, X., SEN, S., KURNIAWAN, D., GUNAWI, H., AND JIANG, J. E2e: embracing user heterogeneity to improve quality of experience on the web. In *Proceedings of the ACM Special Interest Group on Data Communication*. 2019, pp. 289–302.

[146] ZHANG, Y., AND DUFFIELD, N. On the constancy of internet path properties. In *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement* (2001), ACM, pp. 197–211.

[147] ZHU, Y., LIU, J., GUO, M., BAO, Y., MA, W., LIU, Z., SONG, K., AND YANG, Y. Bestconfig: tapping the performance potential of systems via automatic configuration tuning. In *Proceedings of the 2017 Symposium on Cloud Computing* (2017), ACM, pp. 338–350.

Knob	Function
TargetAlgo	Sets corresponding tuning algorithm.
TargetNC	Controls the clustering strategy for NCs.
init_samples	Number of samples to initialize an NC.
NCSpread	Controls the performance spread that bounds a NC, and hence the number of cluster (K discussed in § 3.2).
AllowedConfig	Limits the space to disallow certain configurations.
PerfMemory	Length of history for configuration's performance over time.
UpdateLatency	Set latency b/w central *Config. Manager* and servers.
UpdateFreq	Controls the time after which a model is updated.
ChunkSize	Controls the time window for goodput, RTT and loss-rate measurements from packet traces.

Table 4: Simulator knobs

A Fingerprinting Configurations

Our fingerprinting techniques are inspired from recent works [49, 92, 110, 143]. Our tool inter-operates with TLS and infers configurations in the following ways: (i) HTTP configurations are visible to client during the connection setup and are fingerprinted from the server response, (ii) TCP configurations like RWIN are scraped from the packet headers, (iii) TCP initRTO is measured by emulating a loss during TCP handshake (i.e., by not acknowledging SYN packet back to the server), and measuring the time it takes the server to retransmit the SYN/ACKs, (iv) For TCP ICW, a big enough object URL is scraped from a website, the corresponding object is fetched and the number of packets sent by the server in first RTT is measured. Further, we use MSS=64B to trigger higher number of packets from server. We used AWS in respective regions as the vantage points for fingerprinting the configurations.
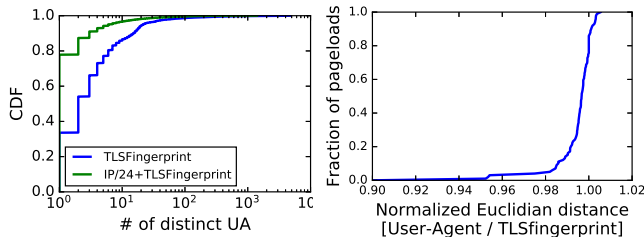
Figure 15:
Relationship between TLS
fingerprint and User-Agent.
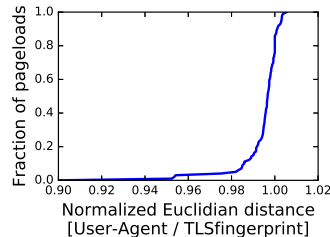
Figure 16:
Comparison of device
identifier's impact on NCs.

B TLS Fingerprinting for Device Identification

Recall that instead of the traditional User-Agent string, Configanator uses TLS fingerprinting for device identification as it allows device inference in early stages of the connection (prior to the HTTP version negotiation through ALPN). To evaluate its efficacy, we leverage a dataset from GlobalCDN, comprising 3.6M requests. The dataset consists of server logs and captures User-Agent strings from HTTP GET requests and the TLS fingerprint of the respective connections. The

dataset includes 14.5K unique User-Agent strings and 3.2K unique TLS fingerprints.

Figure 15 plots the number of unique User-Agents (UA) that map to a TLS fingerprint. Ideally, a single UA should map to a fingerprint, thereby accurately identifying the corresponding device. However in practice, we observe that the one-to-one mapping is limited only to 34% of the fingerprints, with the rest mapping to atleast 2 UA. We observe that complementing the TLS fingerprint with the end-user IP-prefix helps in improving the accuracy, with 78% of the IP/24 and TLS fingerprint mapping to a single UA and 96% mapping to at-most 8 unique UA. We observe that for the cases where a single fingerprint maps to multiple UA strings, there are only minor differences, e.g., different browser versions, difference in OS's minor version (Android 6.0 vs 6.1.1).

In Figure 16, we further compare the two device identification techniques for clustering similar connections together. Using a dataset of 89K PLT measurements from GlobalCDN, we run our Network Class clustering using either User-Agent or TLS fingerprint as the basis for device identification. We compute the Euclidean distance of each connection PLT from its cluster's center and the figure plots the ratio of the distance. We observe that the ratio is between 0.98 and 1.00 for the overwhelming majority of the pageloads, indicating that the two technique perform fairly similar. Hence, device identification through TLS fingerprinting provides nearly similar accuracy to the User-Agent strings, with the added benefit that the device is identified prior to negotiating the HTTP version, whereas User-Agent string can only be inferred through HTTP requests headers (received after HTTP version negotiation).

C Passively Recording Network Conditions

Configanator passively collects goodput and packet loss rates for the IP-prefix (/24) and builds a historical archive (§ 3.2). When an IP connects, Configanator uses the handshake RTT and looks-up the goodput and packet loss rates from the recent session for the IP-prefix (/24) to aid in classifying the user into her Network Class. Configanator prototype uses Apache logs to collect information about user IP, the requested content, content size and download time. Additionally, per-connection TCP statistics are captured through Apache TCP Info plugin [117]. Using this information, Configanator calculates bandwidth (goodput) and packet loss rates on a per IP-prefix (/24) basis. Although we use heuristics for stable goodput and loss calculations, e.g., ignoring small objects, the goodput estimate may still under-estimate actual network bottleneck due to TCP mechanics (e.g., slow start phase). Consequently, the use of such measurements in the testbed (§ 6.1) may emulate lower bandwidths and higher loss rates (emulated loss plus induced buffer overflows) than the actual bottleneck links.
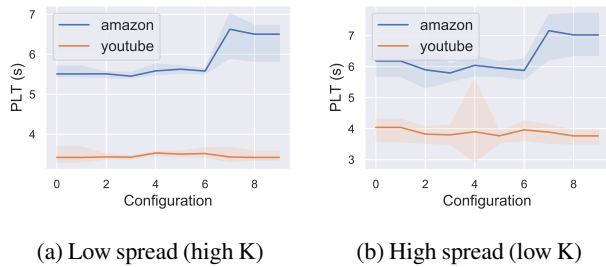
(a) Low spread (high K)   (b) High spread (low K)

Figure 17: GP config-performance curve.

D   Gaussian Process and Network Class Discussion

Bootstrapping GP: The first step of learning is to acquire data to bootstrap the Gaussian process. The bootstrap methodology is crucial for ensuring that the Gaussian-Bandit quickly finds good direction to explore. Recent works [4, 13, 32] have demonstrated the applicability of three distinct bootstrapping approaches: (i) *random*, in which the initial configurations are randomly selected; (ii) *domain-specific*, in which prior domain knowledge, captured through operator interviews or offline simulations, are used to rank configurations to sample; (iii) *Latin Hypercube Sampling* (LHS) which divides the input space into partitions and selects a sample from each partition to spread the samples evenly across space [122]. In this work, we use LHS to bootstrap the learning process. LHS has been found to aid bootstrapping Bayesian optimization by reaching an optimal decision quicker [86]. We observed LHS to speed up exploration in comparison with others by reducing the number of optimization steps by 2-3X, as the bootstrapping samples are spread evenly across space. A perfect rankings of configurations cannot be known prior to actually testing configurations, leading to ranking-based bootstrapping being sub-optimal to LHS.

Individual GP models for each website/Network Class: Bayesian Optimization is traditionally used for mapping configurations to their performance per workload (e.g., cloud configuration to cost [4]). Due to network dynamics and their implications on web performance [67, 135], a separate BO/GP model is required to map configuration performance for each workload (network condition and website), leading to individual exploration for each workload. The lack of cross network/website exploitation (due to separate BO models) makes a solely BO-based technique unfit for Configanator. Intuitively, the system should be able to generalize across networks and can use the already learnt pattern from other networks to a new network, e.g., HTTP/1.1 is optimal at high RTT, high loss for a complex website, no matter the bandwidth [135].

Figure 17 presents the GP model for two websites for the same set of configurations (x-axis) and the same Network Class (NC). In Figure 17a, while GP has estimated the curve for *youtube* with high confidence, *amazon* requires more data samples (wide confidence interval for configuration 7, 8 and

9). The different configuration-performance curves require a separate GP model for each website/NC for correct modeling of a configuration's performance and effective exploration, as the configuration-performance curve is distinct for every website and Network Class.
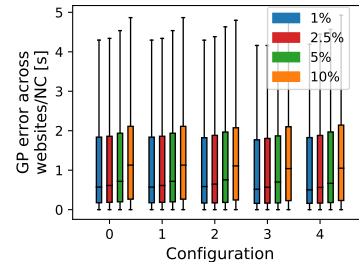
Figure 18: GP modeling error for different NC spread thresholds.

Impact of performance spread within NC: Next, in Figure 18, we leverage the *NCSpread* knob in simulator (Table 4) to test different bounds for Network Classes (NC) clustering. Recall that *NCSpread* controls the "K" for Kmeans clustering by selecting the lowest K that bounds the standard deviation of PLTs for a cluster's constituents within a specified threshold ({1, 2.5, 5, 10}% in the Figure 18). Determining the right K involves iterating through K values and is a three step process: (i) NC features – network characteristics (bandwidth, latency, loss rate), AS information (ASN, geo-location), and device type – from past connections are clustered using a given K, (ii) For each cluster, the list of PLTs observed for its members connections is generated and is normalized by the median PLT of the list[11], (iii) The standard deviation for each list is computed and, based on how far is it from the median and the *NCSpread* limit, the decision to converge on the given K or test a different K is made.

Figure 18 uses the testbed generated data from § 6 and plots the error in GP's estimate for five randomly selected configurations. The error is calculated as the absolute difference of GP's PLT estimate for a configuration and the actual PLT, and the boxes plot the error distribution observed for the various clusters (corresponding to the *NCSpread* value) and the websites. Note that, a small error is always expected due to the inherent variability with PLT measurements. The 5% limit *NCSpread* performs fairly close to the lower bounds, while also requiring a lower K: 7% lower K value than the 1% *NCSpread* threshold. This analysis serves as the motivation for using 5% value in the simulator.

Figures 17a and 17b further visualizes the confidence intervals for the GP models for 2 websites. For the high spread case (10% *NCSpread*), connections from slightly different

---

[11]As there might be multiple websites, there is one list per website. Further only PLTs for default configuration are used.

networks are mapped to the name GP, resulting in wider confidence intervals, and leads to inefficiency with acquisition function's next configuration suggestion.

E Deployment Considerations

Data-driven systems [2, 46, 66–68] traditionally use a split-plane architecture where a modeling layer (responsible for ingesting huge amounts of data and updating models) runs at a slower granularity than the decision layer (responsible for applying modeled decisions for users at real-time). Configanator's architecture uses a split-plane model which leverages the different computational requirements of Configanator's workflow: As demonstrated in Figure 4, in the slow path, the Configuration Manager collects telemetry from the web servers, uses this telemetry to update the learning model, and installs the configuration rules created by the model into the web servers. In the foreground, each web server uses the pre-installed rules to apply configuration to each connection and periodically collects telemetry from each connection.

The *first* phase, the background process, is time-consuming because of the process of updating the learning algorithms and Network Classes. The *second* phase, a fast, real-time process that applies the configuration rules to each inbound *user connection*, is run at the edge on each web server and provides low-latency, dynamic tuning. We note that although this decoupling results in the fast-path using stale information, we observe that this stale information still provides near-optimal performance [46].

F Supplementary Evaluation Material
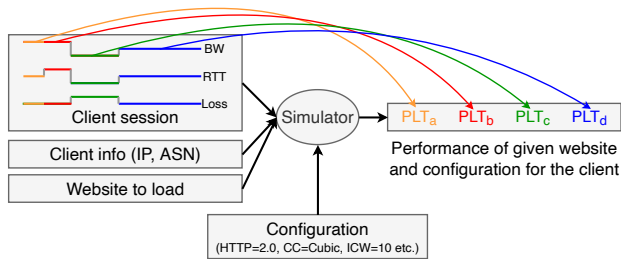
Figure 19: Simulating pageload for a client

F.1 Evaluation Setup

Simulation workflow: Figure 19 presents the workflow for simulating pageload performance. The client sessions are extracted from the real-world datasets and are modeled as time-series. Since we use 5s for measuring the network characteristics from the trace (a tunable knob as discussed in § 6.1), each linear state for BW, RTT, Loss in Figure 19 is at

least 5s long. We extract an IP distribution from the trace to model the temporal aspects of client's connections (time at which a client connection (or IP) is seen in trace), i.e., the user sessions are fed to the simulator in the order they are observed in the real-world trace.

The simulator takes the goodput, RTT, loss rates at a certain time from the session time-series, client info (IP, ASN) and the target website to load as input. Using these features, it consults the configuration to test from the learning framework. Once the target configuration is known, it leverages the PLT-Tensor to map the network characteristics {goodput, RTT, loss-rate}, website and configuration to the eventual PLT. Note that, we assume that the network characteristics stay stable throughout the lifetime on a single pageload, supported by recent studies that TCP connection is piece-wise stationary and each segment stays stable in the order of tens of seconds to minutes [10].

Table 4 further summarizes a number of simulator knobs that allow us to emulate and test a variety of scenarios.

Dataset description and breakdown: While the Global-CDN, MAWI and CAIDA datasets are adequately described in § 6.1, here we provide details for the other two datasets.

The Pantheon dataset [141, 142] comprises of synthetically generated TCP flows across the different parts of the world. We collected three month's worth of data (May to July 2018) from Pantheon's website [141]. For the generated flows, the dataset logs the flow IDs, packet ingress/egress timestamps, packet sizes and one-way delay. Using these fields, we calculate the goodput, RTT and loss rates between each pair of end-point and, similar to the case for GlobalCDN, MAWI and CAIDA datasets, generate the time-series for the network characteristics. These end-points (vantage points) range from AWS deployments to university networks and cover multiple last-mile connection types. The FCC dataset is collected by the Measuring Broadband America program [41] and consists of a nation-wide study of end-user's broadband performance and an accompanying dataset. This dataset provides coarse granularity measurements in form of bandwidth, latency and loss rates distributions measured for real-world users. We use these distributions to generate synthetic traces, similar to [2].

The breakdown of ∼21.4M sessions is as follows: 8.2M from GlobalCDN, 2.7M from MAWI, 8.1M from CAIDA, 1.6M from FCC, 800K from Pantheon. The cross-regional nature of our datasets provide coverage over a wide range of representative network conditions, e.g., while FCC and CAIDA cover connections in U.S., MAWI dataset is from East Asia. Further, GlobalCDN and Pantheon [142] are even more diverse with connections from countries across the globe.

SpeedChecker and vantage points: SpeedChecker [81] is a platform for global Internet measurements, with vantage points deployed in over 170 countries and thousands of ISPs. SpeedChecker provides an API to conduct automated measurements ranging from ping, DNS, web pageloads to video tests. We leveraged vantage points (windows machines) on this platform for conducting the pageloads. The API call

(a) Network at vantage points (b) # of pageloads per prefix
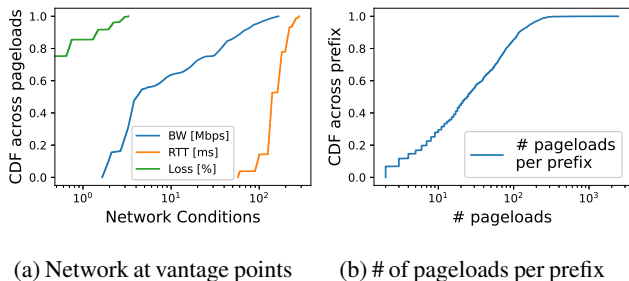
Figure 20: Live-deployment vantage points

requires *CountryCode* and *Destinations* (a list of URLs to load). Vantage points (probes) from the specified country are selected internally by their platform and pageloads are conducted (upto 100 pageload every hour, per city in the country). Figures 20 presents various distributions about our vantage points. 20a compiles the distribution of network conditions observed for each pageload. The vantage points vary across the three dimensions and have mostly RTTs greater than 100ms. 20b presents the number of pageloads per prefix. We observe a heavy tail distribution, where certain vantage points conducted more pageloads than others, e.g., Europe, N.America had 4X more pageloads than Asia and Africa due to the higher number of the SpeedChecker clients in the developed regions. Africa had the smallest number of vantage points among all continents and the hourly limits were frequently reached, resulting in a lower number of total pageloads. Note that, diverse network conditions were still observed for the vantage points (Figure 20a) in spite of this skew in vantage point location. We further observe that 90% of the vantage points have unique IP-prefix (/24), showing that they are distributed and are not placed in a single facility, in the same subnet.
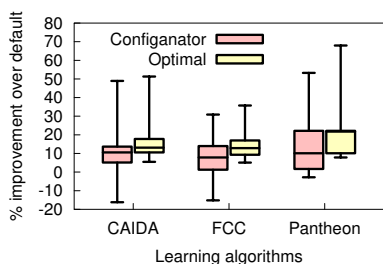
Figure 21: Configanator performance for CAIDA, FCC and Pantheon traces

F.2 Configanator Performance for CAIDA, FCC and Pantheon Traces

Figure 21 presents the distributions for Configanator's PLT improvement for the CAIDA, FCC and Pantheon traces. These figures complement the results in § 6.2 where we could not add

the results for all the traces due to space limitations. CAIDA and FCC traces are collected from U.S.A and mostly cover high bandwidth, low RTT/loss connections, e.g., p95 RTT is 60ms. Following the trend observed in Figure 5c, we observe their PLT improvements over default to be lower as compared to other traces. Especially FCC dataset covers broadband connections and we observe the lowest p95 PLT improvement for FCC among all the datasets. Nevertheless, the improvements are still substantial with 610-640ms decrease in p95 PLT. On the other hand, Pantheon traces cover wider range of networks, often across continents, and result in upto 850ms improvement at tail.
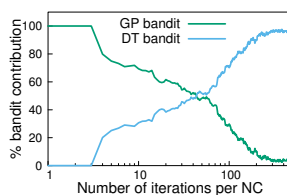
Figure 22: Bandits contribution
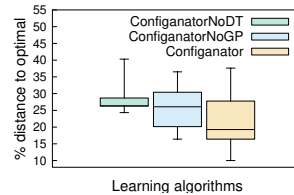
Figure 23: Tail performance

F.3 Bandit Contribution

Figure 22 uses the same convergence analysis as Figure 6 and plots the percentage of connections that uses a certain bandit. Initially GP bandit is largely used for a guided exploration. However, as more data is collected, DT bandit starts to overshadow the GP bandit, highlighting that a per-NC guided exploration is over-shadowed by cross-NC exploitation, when large data is available.

F.4 Bandit Performance at Tail

Figure 23 focuses on tail by dividing the entire trace into one minute segments and plotting the distance to optimal for the worst-case tail of each minute. Configanator's use of bandits enables it to perform better than individual bandits, being closer to optimal by more than 7%.
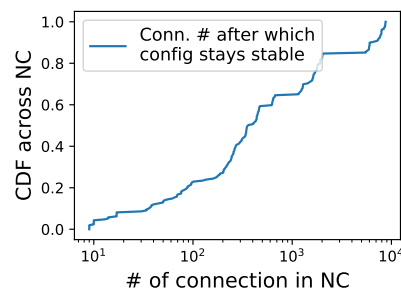

Figure 24: Time of last config. change in NC

F.5 Configuration Stability

Figure 24 plots the number of connections across NCs after which the DT-bandit's decision stays stable, i.e., configuration decision for the NC does not change. While for the median NC, the configuration choice becomes stable at ∼400th connection; we observe that it can take as much as 10K connections to reach the final configuration for some NCs. We observe the DT-bandit to stuck on a near-optimal configuration for these NCs. Down the line, the epsilon-bandit, randomly exploring, finds the optimal configuration and updates the NC. We note that Configanator switches to DT-bandit in the first 10-15 iterations for these NC, highlighting that the GP model's EI threshold was reached very early, and the initial exploration through GP was not very beneficial in uncovering the optimal configuration.

F.6 Design Choices for Network Classes

We use *GlobalCDN* dataset to evaluate design choices for classifying similar users together. We compare Configanator's clustering with: (i) *IP-Prefix* clusters /24 users together, (ii) *Hobbit* [74] improves /24 groups by merging dis-contiguous /24s based on co-location in Internet topology and homogeneous performance, (iii) *Latency Driven* inspired from AP-Atoms [105] where users with similar latency are grouped together, (iv) since CDNs group users based on their performance similarity [26, 119], *CDN Aggregation* use the natural CDN grouping and assigns all users mapped to a PoP to the same NC. We extract these mappings from the GlobalCDN dataset and, as these mappings can vary over time, build a time-series of user to CDN PoP mapping.
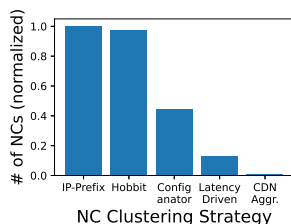
Figure 25: Impact of clustering on # of NC
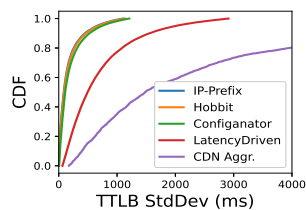
Figure 26: Spread of TTLBs within NC

Figure 25 and 26 plots the number of NCs and the spread of TTLBs within an NC for the different strategies. Ideally, Configanator favors small number of NCs and aims for small to negligible variations within NC performance metric, as the goal is to cluster similarly performing users together (§ 3.2). We observe *Hobbit* subnets /24 groups to have a poor coverage over the trace (Hobbit only covers 12% of prefixes in Global-CDN dataset), with non-Hobbit /24s being treated as individual groups, leading to similar results as *IP-Prefix* (figure 25). Although NCs built by *Hobbit* and *IP-Prefix* have lowest performance divergence, (low std. dev. in figure 26); Configanator NCs are almost similarly compact, while using less than half
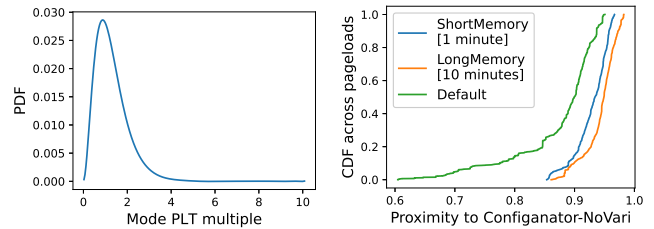
(a) Observed PLT variations.   (b) Impact of PLT variability.

Figure 27: PLT variability

number of NCs. Although *Latency Driven* uses least number of NCs, the lack of device, bandwidth, loss etc. information leads to diverse users being grouped together (high TTLB std. dev.). Similarly, since CDNs maps user based on latency to their closest PoPs, network and device heterogeneity still exist (e.g., the closest PoP to a user can be 10ms-600ms [26]), leading to highest performance variation within an NC for *CDN Aggr.*

We further modified the simulator to explore Configanator performance when different NC techniques are used. We observe that Configanator out-performs the rest for the majority of the pageloads. The prefix and CDN based approaches either do not account for network dynamics or overfit to specific regions respectively. Latency driven performs slightly better but ignoring the important metrics, like packet loss and bandwidth, degrades its effectiveness.

F.7 PLT Variability

PLT measurements are inherently noisy [96] and the variability in PLT can disrupt the learning algorithm's model, e.g., GP is sensitive to noise [78]. Using data from a web performance observability company (NewRelic [71]), we modeled PLT variability distribution and used it to introduce variability in testbed-generated PLT-Tensor. Figure 27a plots a PDF of the variations with x-axis as the mode PLT multiple (x-axis is PLT normalized by mode PLT). We fit an Erlang curve to the observed PDF, owing to its right-skewed, long-tail nature. Using PLT from PLT-Tensor as the mode PLT (since mode PLT is the most stable PLT measurement), the PDF is used to calculate the noisy PLT observed by a real-world user.

Figure 27b plots the extent to which Configanator decisions (in face of PLT noise) are optimal, compared to the case when there is no noise (Configanator-NoVari). A proximity score of 1 indicates that Configanator decisions stay exactly the same for both (noise, no-noise) cases. Leveraging the *PerfMemory* knob, we test 2 scenarios with different length of historical memory. Configanator uses this historical memory to amortize the impact of any sudden change in performance metrics. We observe Configanator's decisions to slightly deteriorate in face of noise. However the extent is mild at worst — with the system still assigning the optimal decisions more than 95% at median.
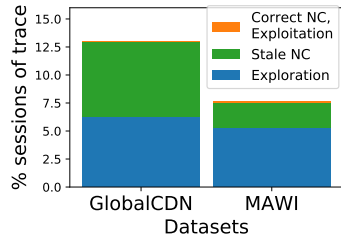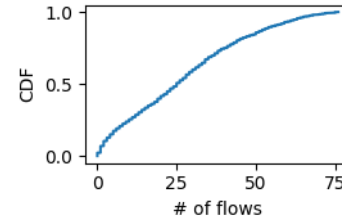
Figure 28: % sessions with PLT degradation.

Figure 29: Number of flows through access link

F.8 Dissecting PLT Degradation

As shown in Figure 5, all algorithms result in some PLT degradation. Figure 28 plots the percentage of sessions that faced PLT degradation and further divide them into the root-causes. Our observations are as follows: (i) During exploration, multiple configurations are tested and may result in degradation. Around 5-6.2% of the sessions in two of the datasets are such exploration steps. (ii) As network conditions change over time, Configanator's estimate of historical network characteristics for an IP-prefix may diverge from the actual network. The stale information is used for classifying the connection into an NC and predicting the optimal configuration. Due to the global nature of GlobalCDN dataset, we observe a higher network churn, with 6.6% of total sessions resulting in PLT degradation due to stale NC. Only a small proportion of sessions in MAWI dataset (∼0.1%) resulted in PLT degradation with correct NC view, indicating that the exploitation arm momentary got stuck at a sub-optimal configuration.

F.9 CM Design Choices

Configuration Manager Design: CM can run locally in a PoP or centrally within a data-center [46], trading-off between data-size to learn and the speed to react to changes. We evaluate both scenarios in our simulator: In the local design, there's a separate CM for each trace, while for the global case there's a single CM for all traces. To simulate each scenario, we vary the latencies between CM and the web servers. We observe that while the global CM is able to make slightly better predictions at the tail (2% better than local), the difference at median is

---

[12] It takes ∼2 minutes to update the models for 10K sessions.

negligible. Despite the larger data set, global CM is not significantly better due to distinctly diverse network conditions across regions (only 17% NCs are common in U.S and Japan traces).

Frequency of model updates: Next, we analyze the impact of updating our performance model less frequently: we explore a range of values from every 2 minutes [12] up to every day. We observed performance to stay relatively stable at the median, whereas hourly or lower update intervals result in ∼8% better improvement at tail, than a per-day granularity.

F.10 Flows Through Access Link

We use packet trace from [115] to measure the typical number of TCP flows through an access link. Figure 29 presents the number of TCP flows with at least 10Kb data transferred, in a 60s time interval. On the median 60s time interval, we observe around 25-30 flows competing through the access link.

F.11 Additional Micro-benchmarks

In addition to the system benchmarks in § F.9, we also evaluated two alternate design choices: VMs and LD_Preload. For VMs, we used one VM for each configuration and used Open vSwitch (OVS) [43] for routing flows to the appropriately configured VM. We explored the use of *LD_Preload* to intercept system call and tuned socket using *setsockopt()*. In comparing both choices with Configanator, we observed that the VM-based approach introduced a 20% increase in latency where as the *LD_Preload* introduced a much smaller latency of 2.2%. We also observed overheads for CPU and Memory utilization: the VM-based approach introduced 30% (memory taken by the guest OS) while *LD_Preload* introduced a 5% increase.

C2DN: How to Harness Erasure Codes at the Edge for Efficient Content Delivery

Juncheng Yang[1], Anirudh Sabnis[2], Daniel S. Berger[3], K. V. Rashmi[1], Ramesh K. Sitaraman[2,4]

[1]Carnegie Mellon University
[2]University of Massachusetts, Amherst
[3]Microsoft Research and University of Washington
[4]Akamai Technologies

Abstract

Content Delivery Networks (CDNs) deliver much of the world's web and video content to users from thousands of clusters deployed at the "edges" of the Internet. Maintaining consistent performance in this large distributed system is challenging. Through analysis of month-long logs from over 2000 clusters of a large CDN, we study the patterns of server unavailability. For a CDN with no redundancy, each server unavailability causes a sudden loss in performance as the objects previously cached on that server are not accessible, which leads to a miss ratio spike. The state-of-the-art mitigation technique used by large CDNs is to replicate objects across multiple servers within a cluster. We find that although replication reduces miss ratio spikes, spikes remain a performance challenge. We present C2DN, the first CDN design that achieves a lower miss ratio, higher availability, higher resource efficiency, and close-to-perfect write load balancing. The core of our design is to introduce erasure coding into the CDN architecture and use the parity chunks to re-balance the write load across servers. We implement C2DN on top of open-source production software and demonstrate that compared to replication-based CDNs, C2DN obtains 11% lower byte miss ratio, eliminates unavailability-induced miss ratio spikes, and reduces write load imbalance by 99%.

1 Introduction

Content Delivery Networks (CDNs) [20] carry more than 70% of Internet traffic and continue to grow [19]. Large CDNs achieve this by operating thousands of clusters deployed worldwide so that users can download content with low network latency. When a user requests an object, the CDN routes the request to a server proximal to the user [15]. If the server contains the requested object in its cache, the user experiences a fast response (*cache hit*). If no server within the cluster has the object in cache (*cache miss*), the object is fetched from a remote cluster which could be another CDN cluster or the origin (i.e., the content provider).

Detrimental effects of cache misses. Cache misses have three detrimental effects. First, they degrade performance by increasing the *content download times* experienced by the user, as each object incurring a cache miss would have to be downloaded over the WAN from a remote server. Second, cache miss can result in additional traffic between the CDN cluster and the origin, which is a significant bandwidth cost for CDN operators. Third, if more cache misses are served from the origin, content providers need to provision more servers with higher network bandwidth. Consequently, a CDN's goal is to minimize the miss ratio and maintain a low miss ratio over time for all content providers.

Why tail performance matters. The design goal of a CDN is to consistently improve download performance for *all* objects on a content provider's site, in *every* time window, and for *each* client location. The performance improvement is viewed as a "speedup" that the CDN provides over the content provider's origin, i.e., it can be quantified as the ratio of the time to download an object directly from origin (without the CDN) to the time to download the same object from the CDN. A CDN's goal is to provide a significant average speedup in every time window (say, 5-minute window) and at each client location. A spike in the miss ratio in a single cluster could violate these performance goals, *even if that spike is short-lived and impacts only a subset of the objects*. That is because the CDN likely offers no speedup over origin for any client download that is a cache miss, and indeed a short-lived spike in miss ratio could drastically decrease the average speedup provided by the CDN during a 5-minute period.

The challenge of frequent server unavailabilities. Due to stringent performance goals, servers are continuously monitored by the cluster's load balancer. A server is declared to be "unavailable" and (temporarily) taken out of service if it is deemed incapable of serving content to users within specified performance bounds. By analyzing a month long logs from the load balancers in over 2000 clusters of a large CDN, we find that server unavailability is very common in CDN edge clusters. When a server is unavailable, the objects stored in its cache are not available to serve user requests. Unless the requested objects can be retrieved from other servers within the

cluster, these objects need to be fetched from a remote server, resulting in a spike in the miss ratio, potentially causing a violation of performance guarantees.

Limitations of the state-of-the-art approaches. To tolerate server unavailabilities, the state-of-the-art approach adopted by large CDNs is to replicate objects across two servers within a cluster. We found that this approach has three significant limitations. First, we find that object replication does not eliminate the miss ratio spike following a server unavailability event. The reason is that the replica of the object (within the cluster) may no longer be present due to eviction from its cache. Second, replicating objects is space-inefficient as the CDN effectively has to provision twice the cache capacity, which is challenging due to the accelerating growth in CDN traffic. Third, we observe a significant *write* imbalance between servers due to DNS based load balancing. This imbalance increases SSD read latency and reduces SSD lifetime [22, 61].

Bringing together efficiency and high availability. In this paper, we present C2DN[1], a CDN design that achieves both high availability and high resource efficiency. To achieve high resource efficiency, we apply erasure coding to large cached objects. This requires overcoming multiple CDN-specific challenges such as eviction of object chunks due to write rate imbalances. In fact, we show that a naive application of erasure coding fails to achieve the goal. The core of our design is a new technique that enables CDNs to balance eviction rates and write loads across servers in each cluster. We exploit the fact that erasure coding enables more flexibility in assigning chunks to multiple servers. Our key insight here is that the chunk assignment can be reduced to a known mathematical optimization problem, called Max Flow Problem.

The core contributions of C2DN are a novel chunk placement scheme for consistent-hashing-based load balancing in CDN clusters and a low-overhead implementation of erasure coding for CDNs that can serve the different traffic requirements of production systems. Specifically, by solving an instance of the Max Flow problem, we assign objects with *near-optimal balance* in eviction and write rates for CDN servers and their SSDs. As a consequence, C2DN can reduce storage overheads and bandwidth costs. Finally, equal write rates across servers essentially function as a cluster-wide distributed wear-leveling for the servers' SSDs, significantly extending lifetimes.

Our contributions. We make the following contributions.

1. We show that server unavailability is common in CDN clusters by analyzing a month-long trace from over 2000 load balancers of a large CDN. We show that the state-of-the-art approach of replicating objects within a cluster does not eliminate miss ratio spikes after a server unavailability events.

2. We design C2DN with a hybrid redundancy scheme using replication and erasure coding, along with a novel approach for parity placement. C2DN reduces the storage overhead of providing fault tolerance, and hence lowers the miss ratio. Moreover, by leveraging the parity assignment, C2DN balances the write loads and eviction rates across cache servers.

3. We implement C2DN on top of the Apache Traffic Server (ATS) [7] and evaluate it using production traces. We show that C2DN provides 11% miss ratio reduction compared to the state-of-the-art, and C2DN eliminates the miss ratio spikes caused by server unavailabilities. Further, C2DN decreases write load imbalance between servers by 99%.

2 Background

We describe CDN architecture, performance, and cost factors.

CDN Architecture. A CDN is a large distributed system with hundreds of thousands of servers deployed around the world [20, 50]. The servers are grouped into *clusters*, where each cluster is deployed within a data center on the edge of the Internet. The CDN cluster caches content and serves it on behalf of *content providers*, such as e-commerce sites, entertainment portals, social networks, news sites, media providers, etc. By caching content in server clusters proximal to the end users, a CDN improves performance by providing faster download times for clients. Unlike storage systems, CDN servers do not store the original content copies. When the requested content is not available in the cluster (cache miss), the content is retrieved from other CDN cluster or the origin servers operated by the content provider.

Bucket-based request routing. When a user requests an object, such as a web page or video, the *global load balancer* of the CDN routes the request to a cluster that is proximal to the user [15]. Next, the *local load balancer* within the cluster routes the request to one or more servers within the chosen cluster that can serve the requested object. As an example, in Akamai's CDN, these routing steps are performed as DNS lookups. A content provider CNAMEs its domain name (e.g., for all of its media objects) to a sub-domain whose authoritative DNS server is the CDN's global load balancer. At the global load balancer, this sub-domain is CNAME'd to a cluster-local load balancer that assigns the sub-domain to a cluster server using consistent hashing [43].

CDN request routing stands in contrast to sharding in key-value caches, such as Memcached and Redis, where consistent hashing is often applied at a per-object granularity [49, 81–83]. *In CDNs, load balancing decisions are taken on the granularity of groups of objects called buckets*. Each bucket, in a DNS-based load balancer, correspond to a domain name that is resolved to obtain one or more server IPs that host objects in that bucket. This resolution is computed using consistent hashing. Since the number of buckets is limited in the range of 100s, the computation is performed and cached when a cluster server becomes available or unavailable.

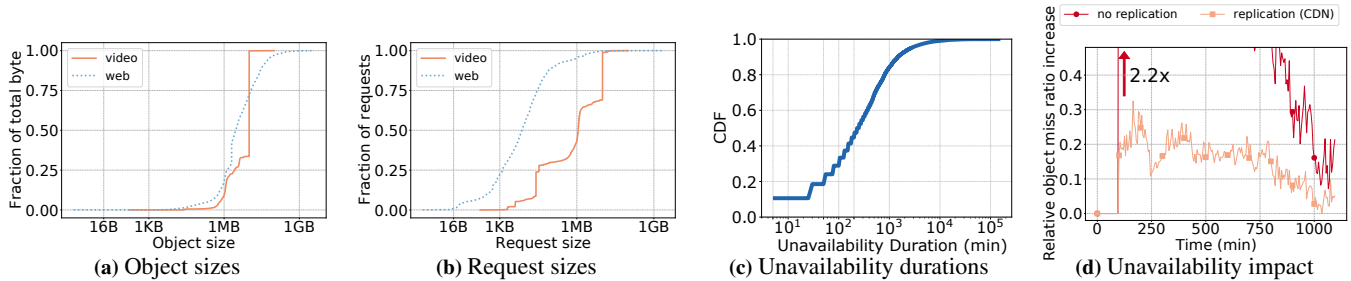
[1] C2DN stands for Coded Content Delivery Network.

Figure 1: a) Size distributions show that large objects contribute to most of the unique bytes and b) most requests are for small objects. c) Server unavailabilities are mostly transient. d) Object miss ratios spike after server unavailability both with and without the state-of-the-art replication.

CDN Performance Requirements. A CDN aims to serve content faster than a customer's origin by a specified speedup factor. This factor is commonly part of a service level agreement (SLA) between the CDN and the content provider. The SLA is monitored by recording download times from a globally distributed set of locations for the same content using both CDN and origin servers. Hence, the goal is to ensure good "tail" performance in *every* time interval for *every* content provider from *every* cluster.

Operating Costs of a CDN. CDNs seek to minimize the operating cost, which consists of the following main categories. (i) *Bandwidth* : A major component of the operating cost of a CDN is bandwidth, accounting for roughly 25% of operating costs. The bandwidth cost can be further broken down, the bandwidth cost caused by cache miss traffic called *midgress* [69] that accounts for roughly 20%, while the rest is the cost of *egress* i.e., the traffic from the CDN servers to clients. CDNs have a great cost incentive to reduce the byte miss ratio and the midgress traffic since a CDN gets paid by content providers for the traffic to end users. The midgress traffic between CDN clusters and the origin is purely a cost overhead for the CDN. Even modest reductions in midgress translate into large cost savings since the bandwidth costs tens of millions of dollars per year for a large CDN [69].

(ii) SSD wearout: A second major cost component is server deprecation which accounts for about 25% of the operating cost of a large CDN. Hardware replacements are particularly expensive for small edge clusters due to the large geographic footprints of CDNs. SSDs are a key component due to the high IOPS requirements of CDN caching. Unfortunately, using SSDs in caching applications is challenging due to their limited write endurance [9, 22, 42, 67, 70]. With deployments of TLC and QLC SSDs, reducing SSD write rates has become even more critical. Besides reducing the average write rate within a cluster, CDNs also seek to reduce the variance of write rates of different servers and their SSDs. Large variance leads to some SSDs not achieving their intended lifetime (e.g., 3 years) as well as high tail latency (see §3.4). Consequently, CDNs seek to reduce the peak write rate, ideally balancing write rates across all SSDs in a cluster.

Per server load (TB)	Max	Min	Mean	Max/min
Weekly read	225.2	167.9	191.2	1.3
Weekly write	16.54	6.69	12.57	2.5

Table 1: Read and write load for a 10-server production cluster.

3   Production CDN Trace Analysis

This section motivates the design of C2DN by analyzing three sets of traces from production Akamai clusters.

We collected request traces from two typical Akamai 10-server-clusters (cluster cache size 40 TB), one mainly serving web traffic and the other mainly serving video traffic. These traces comprise anonymized loglines for every request from every server over a period of 7 and 18 days, respectively. The **web trace** totals 6 billion requests (1.7 PB) for 273 million unique objects (79.8 TB). The **video trace** totals 600 million requests (2.1 PB) for 130 million unique objects (224 TB).

Additionally, we collected availability traces from 2190 Akamai clusters over 31 days. The trace consists of snapshots taken every 5 minutes from each cluster's local load balancer. Each snapshot contains the number of available servers as determined by the load balancer. The smallest cluster has two servers, the largest cluster has over 500 servers, and the median cluster size is 17 servers. We observe that cluster size has a wide range, and around 40% of clusters have fewer than or equal to 10 servers. We plot the distribution of cluster size in Fig. 10 in Appendix 10.1.

3.1   Diversity in workloads and object sizes

CDNs mix different types of traffic in clusters in order to fully use their resources. For example, different "classes" of traffic with small and large object sizes, such as web assets and video-on-demand, are mixed to balance the utilization of the cluster's CPUs as well as network and disk bandwidth [68]. Consequently, object sizes vary widely [10]. Figures 1a and 1b show the size distribution for our production traces, weighted by unique objects and by request count, respectively. As expected, object sizes vary from a few bytes to a few GBs. Fig 1a shows that *the majority of traffic and cache space is used by large objects*. Furthermore, objects smaller than 1 MB make

up less than 15% and 12% of the total working set in web and video, respectively. Fig 1b shows that *the majority of the requests are for small objects* with 95% of requests in web-dominant workload smaller than 1 MB, and 50% of requests in video-dominant workload smaller than 1 MB.

3.2 Unavailability is common and transient

Unavailability is common. Across all clusters, server unavailabilities occur in 45.2% of the 5-minute snapshots. For clusters with only ten servers (same size as the cluster we collect request traces from), we observe that 30.5% of 5-min time snapshots show server unavailability. Moreover, we observe that unavailability affects only a small number of servers at any given time: 85% of unavailabilities affect less than 10% of servers in large clusters, and 84% of unavailabilities affect no more than a single server in a ten-server cluster.

These unavailability rates can appear high compared to published failure rates in large data centers [25, 46, 54, 56, 59] and HPC-systems [65]. However, environmental conditions can be more challenging in small edge clusters. For example, edge locations often have less efficient cooling systems than highly optimized hyperscale data centers; edge clusters also have less power redundancy, such as redundant battery and generator backups [50]. Moreover, CDN clusters employ a rigorous definition of server unavailability. When a server does not meet the performance requirement, it is deemed as unavailable by the load balancer. These types of unavailability are rarely reported by data centers and HPC systems. Unfortunately, the unavailability logs do not provide a causal breakdown of failure events.

Unavailability is mostly transient. Fig. 1c shows a CDF of the durations of unavailabilities. We observe that unavailabilities can last between 20 minutes and 24 hours with a median duration of 200 minutes. These short unavailabilities are mostly caused by performance degradation, such as unexpected server overload and software issues (e.g., application/kernel bugs or upgrades). Besides, we observe a long tail of unavailability durations, with around 16% exceeding 24 hours and 2% exceeding an entire week. These cases may be related to hardware issues. Qualitatively, our observations are similar to storage systems in the sense that unavailabilities are common and most unavailabilities are not permanent.

3.3 Mitigating unavailability is challenging

Upon detecting an unavailability, the load balancer removes the corresponding server from the consistent hash ring and reassigns their buckets to other servers [43]. We evaluate how a bucket's object miss ratio is affected by unavailability using the video trace. Fig. 1d shows that the object miss ratio in a CDN cluster *without any redundancy* increases by more than 2× relative to no unavailability over the same time period. *This spike disproportionally affects a small group of content providers because of bucket-based routing* (§3.1).

The high latency resulting from cache misses can lead to SLA violations.

The state-of-the-art mitigation technique for server unavailability at large CDNs is *replicating* buckets across two servers[2]. When one server becomes unavailable, requests are routed to the other server, likely to hold the object. Fig. 1d shows that replication reduces the intensity of the miss ratio spike. However, we find that replication does not remove the miss ratio spike. *In contrast to storage systems, where replication guarantees durability, in CDN clusters, servers perform cache evictions independently.* Objects that are admitted to two caches at the same time may be evicted at different times. This is particularly common if the two caches evict objects at very different rates, making replication ineffective. We next discuss why this case is more common than one might expect.

3.4 The need for write load balancing

We measure the read and write load balance across servers in a CDN cluster. To make the analysis independent from eviction decisions, we present the read and write rates based on compulsory misses from the web trace [3]. Table. 1 shows that the server with the highest read load serves 1.3× more traffic than the server with the lowest read load. The server with the highest write load writes around 2.5× more bytes than the server with the lowest write load.

Write load imbalance causes three problems. First, imbalance reduces the effectiveness of replication. A server with a 2.5× higher write rate also has a 2.5× higher eviction rate. So, a newly admitted object will traverse the cache with the highest write load 2.5× faster than the one with the least write load. Consequently, buckets mapped to these servers will have many objects for which only a single copy is cached in the cluster. We find that for 25% of objects, only a single copy exists in the cluster, which leads to the miss ratio spike observed during unavailabilities (Fig. 1d). Second, SSD write load imbalance often causes high tail latency. Specifically, high write rates frequently trigger garbage collection, which can delay subsequent reads by tens of milliseconds [11, 77, 78, 80]. These delays are significant enough to have been recognized as a problem by multiple CDN operators [61]. Third, the imbalance can lead to short SSD lifetimes due to concentrated writes on some SSDs, and thus higher replacement rates [9, 22], which increases CDN cost (§2).

4 C2DN System Design

C2DN's design goals are to: (1) eliminate miss ratio spikes

---

[2] For operational flexibility, CDNs do not replicate servers as primary/backup. CDNs implement replication using additional virtual nodes for a bucket on the consistent hash ring [36, 47].

[3] Compulsory misses are cache admissions forced by objects not previously seen in the trace (underestimating the real miss and write rate). However, more compulsory misses only lead to more writes and evictions. Therefore, write rates are often proportional to compulsory misses.

caused by server unavailability, and (2) balance write loads across servers in the cluster. Erasure coding is a promising tool to improve availability under server unavailability. We first describe a naive implementation, called C2DN-NoRebal, based on a straightforward application of erasure coding (§4.1). C2DN-NoRebal fails to achieve the targeted goals, and we identify write and eviction imbalance as the key challenge. We then describe a new technique to overcome this challenge (§4.2) that exploits the unique aspects of the use of erasure coding in the context of CDNs.

4.1 Erasure coding and C2DN-NoRebal

Erasure coding is widely used in production storage systems for providing *high availability* with *low resource overhead* [32, 35, 48, 48, 54, 56]. Conceptually, erasure coding an object involves dividing the object into *K data chunks* and creating *P* parity chunks, which are mathematical functions of the data chunks. Such a scheme, called a (K, P) coding scheme, enables the system to decode the full object from any *K* out of the $K + P$ chunks. Thus, caching $K + P$ chunks on different servers provides tolerance to *P* server unavailabilities. As individual chunks are only a fraction $1/K$ of the original object's size, coding reduces space overhead compared to replicating full objects[4].

As CDNs use bucket-based routing (§2), coding needs to be applied at the level of buckets rather than objects. Specifically, the *K* data chunks of all the objects belonging to a bucket are grouped into *K* distinct *data buckets* respectively. Similarly, the corresponding *P* parity chunks are grouped into *P* distinct *parity buckets*. These buckets (data and parity) are each assigned to a distinct server in the cluster. Note that while the routing happens at the level of buckets, requests are still served at the level of objects. Hence we will use the term *buckets* in the context of assignment and *chunks* in the context of serving specific objects.

The application of erasure coding to CDNs is shown in Fig. 2a. To serve a user request, a server reads one chunk from the local cache and at least $K - 1$ chunks from other servers to reconstruct the requested object. To find the location of data and parity chunks, *C2DN-NoRebal* relies on a simple extension of bucket-based consistent hashing. The location of the first chunk is the server the bucket containing the object hashes to. Then, subsequent $K + P - 1$ chunks are read from the subsequent $K + P - 1$ servers on the consistent hash ring.

Owing to the reduced storage overhead, C2DN-NoRebal provides cost benefits by reducing the average byte miss ratio when compared to replication (as seen in our experiments in §6). However, C2DN-NoRebal *fails to eliminate the object miss ratio spike during unavailability* (§6). Specifically, we find that coded caches are even more sensitive to write load imbalance than replication. For replication, eviction rate imbal-

---

[4]The space overhead of an (K, P) coding scheme is $\frac{K+P}{K}$. For example, for $K = 3$, $P = 1$, space overhead is $1.33\times$ as opposed to $2\times$ in two-replication.

ance may cause the second (backup) copy to be evicted, which is required when a server becomes unavailable. Whereas for a coded cache, eviction rate imbalance could lead to any of the individual chunks being evicted, which leads to an effect we call *partial hits*: less than *K* chunks of the object are cached in the cluster, and this prohibits the reconstruction of the object. A partial hit only requires fetching the missing chunks, but incurs the same round-trip-time latency as a miss and thus does not provide a speedup. Further, partial hits become even more frequent during server unavailability, thus deeming C2DN-NoRebal less effective.

4.2 Parity rebalance and C2DN

Having identified write imbalance as a key challenge for erasure coding in CDNs, we next show how we exploit parities in overcoming these imbalances. Our main idea is to assign *parity buckets* to servers in a way that mitigates the write load imbalance caused by *data bucket* assignment.

Like the state-of-the-art in CDNs and C2DN-NoRebal, C2DN applies consistent hashing to assign the data buckets (Fig. 2b). We define a server's data write load as the number of bytes written (i.e., admitted) to cache, counting only data buckets. We also define a bucket's parity write load as the bytes written counting only parity buckets. Every server records data write load and each bucket's parity write load since the cluster's last unavailability event. After an unavailability event, parity buckets are reassigned by the load balancer using this information. The load balancer calculates an assignment of parity buckets to servers to balance write load. This assignment is a non-trivial calculation as not every assignment is feasible: parity chunks cannot be assigned to a server that holds a data chunk of the same object. In general, C2DN's parity bucket assignment problem is NP-hard by reduction from the Generalized Assignment Problem [14].

C2DN's parity bucket assignment algorithm. We obtain an approximate solution in polynomial time using a MaxFlow formulation (Fig. 2c). The solution provides us with feasible server assignments for each parity bucket. We empirically observe that by assigning the parity bucket to the least loaded server among the feasible servers, the write load on each server is well balanced. The inputs to the algorithm are:

1. parity write load of bucket *n* (s_n),
2. data write load on server *i* (l_i),
3. total write load on the cluster (W),
4. current assignment of data buckets to servers,
5. available servers in the cluster ($\mathcal{A}$).

The flow graph (Fig. 2c) is constructed using a source-node (S), parity-nodes corresponding to each parity bucket, server-nodes corresponding to each server in the cluster, and a sink-node (T). We add an edge from the source-node (S) to each parity-node *n* with a capacity equal to the bucket's parity write load (s_n). We add edges from parity-nodes to the server-nodes if the corresponding parity bucket can be placed on that
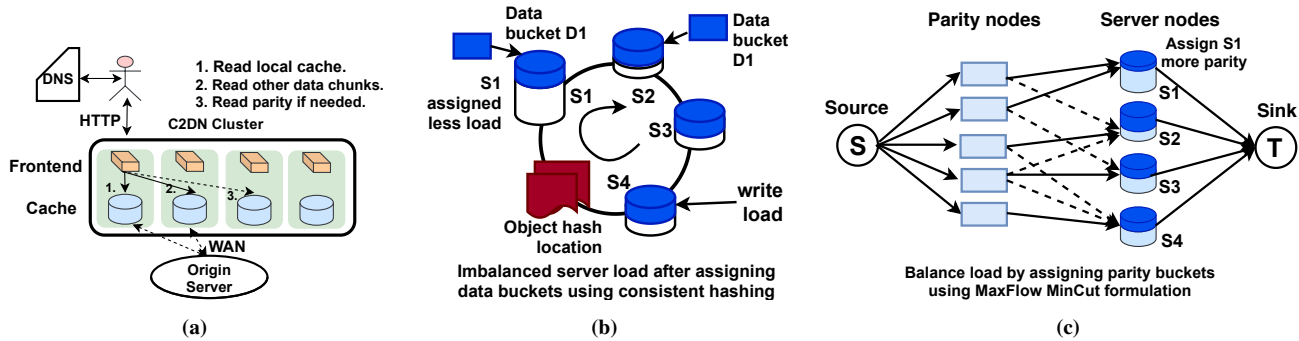
Figure 2: a) **Architecture of C2DN**; b) **C2DN bucket assignment** C2DN assigns data buckets using consistent hashing, which guarantees a consistent mapping across unavailabilities, but causes load imbalance; c) **Parity rebalance.** Write load imbalance is mitigated by assigning parity buckets to balance the load using a MaxFlow formulation.

server, i.e., the data chunks of the bucket are not assigned to the server. The capacity of these edges is again the bucket's parity write load (s_n). Finally, we add edges from server-nodes to the sink-node (T) with a capacity equal to the server's remaining write load budget, which is $\max(\left\lceil \frac{W}{|\mathcal{A}|} \right\rceil - l_i, 0)$.

After solving MaxFlow(S,T), C2DN iterates over parity buckets. Each parity bucket is assigned to the least loaded server with a positive flow from the parity-node to the server-nodes. This leads to a well-balanced assignment. The assignment is also feasible as no positive flow exists between a parity bucket and the servers holding this bucket's data chunks.

The parity rebalance algorithm is described in more detail via a pseudo-code in Appendix 10.3.

Extension to heterogeneous servers. We incorporate heterogeneous servers by setting the capacity of the edge in the graph between server-nodes to sink-node (T) proportional to the size of the server.

4.3 C2DN resolves partial hits

Having shown how to balance write loads across servers within the cluster, we show that this is sufficient to solve the issue of partial hits. Specifically, we find that the probability of a partial hit diminishes for large caches.

We formulate our proof under the simplifying assumptions of the independent reference model (IRM[5]), which is used widely in caching analysis [5, 10, 23]. While our proof can be extended to a range of eviction policies [44], we assume the Least-Recently-Used (LRU) policy for simplicity. We empirically observe that FIFO, which is used in open-source caches such as Apache Trafficserver [7] and our empirical evaluation in §6, behaves similarly to LRU.

We remark that we *do not require explicit coordination of individual eviction decisions among the caches*. Our theorem states that under IRM, in C2DN, if one chunk of an object is present in a cache, then the other chunks are almost surely

---

(a) Decoding throughput   **(b)** SSD throughput

Figure 3: Microbenchmarks. a) With vector instruction in modern CPU, decoding is very efficient with high throughput, the sub-chunk size to achieve maximum throughput across configurations is around 32-64 KB. b) Modern SSD achieves maximum throughput with I/O size larger than 32 KB.

present in the other caches.

Theorem 1. *Under IRM and LRU, in C2DN, for an object with chunks $x_1, \ldots, x_n$, for any $1 \leq i, j \leq n$., and as the cache size grows large*

$$P[\text{chunk } x_i \text{ is in cache} \mid \text{chunk } x_j \text{ is in cache}] \to 1 \quad (1)$$

Our proof uses the fact that balanced write loads lead to equal *characteristic times* [10, 23, 26, 60], which is the time it takes for a newly requested chunk to get evicted from each server's LRU list. Since data and parity chunks of an object are requested simultaneously and the characteristic time is the same, the chunks are also evicted simultaneously, and partial hits become rare. Details can be found in Appendix 10.2.

5 C2DN Implementation

In addition to design goals (1) and (2), C2DN's implementation seeks to (3) minimize storage/ latency/ CPU overheads and (4) remain compatible with existing systems to facilitate deployment. This entails subtle implementation challenges.

Enabling transparent coding. A key architectural question is which system component encodes and decodes objects into/from data and parity chunks. A natural choice might be to encode objects at origin servers. However, this would require changes to thousands of heterogeneous origin software stacks — a barrier to deployment. Additionally, encoding at the origin

---

---

would increase origin traffic as each cache miss needs to fetch both data and parity chunks, e.g., with $K = 3$, $P = 1$ the origin traffic would increase by 33%. Thus, C2DN fetches uncoded objects from origins and encodes chunks within the CDN cluster. Additionally, any decoding operation is also performed within the cluster for transparency on the client side.

Selective erasure coding. While encoding and decoding are fast due to broad CPU support for vector operations, the overhead of fetching becomes significant for small objects. As the majority of requests are for small objects (§3.1), we can reduce processing overheads by using replication for small objects. C2DN applies coding to large objects, which account for most of the production cluster's cache space (§3.1). Of course, with selective coding, we now need to count uncoded objects as part of the data write load in §4.2.

To decide the size threshold of coding, we perform two microbenchmarks studying how coding block size affects coding throughput and SSD bandwidth. Fig. 3a shows that even on a five-year-old Skylake Xeon, decoding is very efficient with per-core throughput over 200 Gbps (data fits in CPU cache) at a block size of 32 KB. This benchmark result suggests that decoding will not be a bottleneck at a reasonable block size (e.g., 32 KB) compared to NIC bandwidth. Fig. 3b shows the relationship between SSD bandwidth and I/O size (setup as in §6). We again find that a block size of 32-64 KB achieves the peak SSD bandwidth. Based on these results, C2DN codes object larger than 128KB so that each chunk is at least 42KB for a (3, 1) coding scheme.

This hybrid approach enables load balancing and space efficiency with no overhead for most requests. One might ask why C2DN relies on replication for small objects after §2 showed that replication continues to suffer from miss ratio spikes. We find that erasure coding large objects is sufficient to balance eviction rates (using C2DN's parity rebalance), making replication effective for small objects.

Parity rebalance and parity look up. As described in § 4.2, C2DN formulates the parity bucket assignment problem as a Max Flow problem. We solve the problem using Google-OR [53], which implements the push-relabel algorithm [18]. The time complexity of this algorithm is $O(n_{node}^2 * \sqrt{n_{edge}})$. where n_{node} is the number of nodes (#buckets + #servers) and n_{edge} is the number of edges ($\approx$#buckets $\times$ #servers). In production systems, #buckets is in the range of 100s for a 10-server cluster. Thus, the time complexity simplifies to $O(\#buckets^3)$. Empirically, we observe low run times as well, for e.g., for 100 buckets and 10 servers, C2DN's parity bucket assignment runs within 50 μs. Also, note that the parity bucket mapping is calculated in the background (off the critical path) and only when there is an unavailability event. From our analysis, we observe around 5.6 unavailability events on an average day.

Support for large file serving, HTTP streaming, and byte-range requests. To minimize latency, CDNs stream large ob-
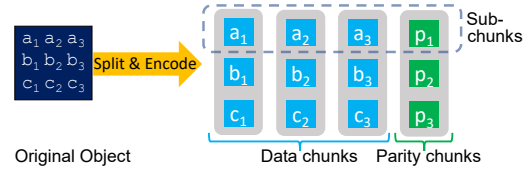

Figure 4: Support for HTTP streaming. C2DN efficiently supports HTTP streaming and byte-range requests by splitting large files into sub-chunks and performs coding on sub-chunks level.

System	Replication(CDN)	C2DN	C2DN reduction
Object miss ratio	0.242	0.227	6.4%
Byte miss ratio	0.118	0.105	11%

Table 2: Object and byte miss ratio from prototype

jects to clients. We achieve compatibility with streaming by subdividing data and parity chunks (for very large objects) into smaller parts which we call sub-chunks. C2DN's encoding and decoding work on the sub-chunk level as shown in Fig. 4. We implement streaming by serving sub-chunks as they become available. For byte-range requests, C2DN fetches the sub-chunks overlapping with the requested byte-range.

Delayed fetch of parity sub-chunks. C2DN can serve a request with any K sub-chunks (out of $K + P$). Because serving with data sub-chunks requires no decoding, C2DN first fetches all K data sub-chunks. C2DN only fetches parity sub-chunks after a heuristic wait period to overcome stragglers. We record the time until the first data sub-chunk is returned. If, after an additional 20% wait time, fewer than K data sub-chunks have arrived, C2DN fetches parity sub-chunks.

Hot object cache (HOC). To facilitate serving hot objects, C2DN caches decoded sub-chunks in DRAM so that if an object is popular, it will be served directly and efficiently from DRAM, thus avoiding fetching and possible decoding.

Metadata lookups. In the case of a HOC miss, C2DN needs to know if the object was encoded or replicated. Storage systems can rely on external metadata for this case, which is not available in CDNs. Thus, C2DN stores metadata with each cached object, indicating whether the object is coded or not. On a HOC miss, C2DN first looks up the object in its *local SSD cache*. If the metadata indicates a coded object, C2DN fetches chunks from other caching servers within the cluster. In the case of a local cache miss, C2DN retrieves the object from other CDN clusters or the origin servers, then C2DN serves the object to the end-user, stores it locally, and encodes or replicates based on the object size.

6 Evaluation

We build C2DN on top of Apache Trafficserver and evaluate it via a series of experiments on Amazon EC2. To study a more comprehensive parameter range, we use simulations. The source code of our prototype and simulator is released at https://github.com/Thesys-lab/C2DN.

The highlights of our evaluation are: (1) C2DN eliminates miss ratio spikes after unavailabilities. Additionally, C2DN re-

duces byte miss ratio by 11%, enabling significant bandwidth cost savings at scale. (2) C2DN reduces write load imbalance by 99%. (3) C2DN achieves the same latency, lower average SSD write rates with only a 14% increase in CPU utilization.

6.1 Experimental methodology and setup

Traces. We evaluate C2DN using the two production traces described in §3. In the following sections, we focus on the video trace and present results for the web trace in §6.7.

Prototype evaluation setup. We emulate a CDN's geographic distribution by placing sets of clients, a 10-server CDN cluster, and an origin data center in different AWS regions. CDN servers use i3en.6xlarge VMs with 80 GB in-memory cache and 10 TB disk cache. To reduce WAN monetary bandwidth costs of the experiments, we measure latency via spatial sampling [72, 73] for 2% of requests. The remaining requests are generated in the same region.

Unless specified otherwise, we use Reed-Solomon codes ($K = 3, P = 1$). We only code objects larger than 128 KB (§4). The prototype experiments use four days of requests to warm up caches. Measurements are then taken for three days of requests. This corresponds to replaying 1.18 PB of traffic in total from local and remote clients in each prototype experiment.

Simulation setup. We implement a request-level cluster simulator. While the simulator does not capture system overheads, it is useful in comparing various schemes for the full duration of the trace and for various cache sizes (which are prohibitively expensive to perform using prototype experiments.) Simulations use 18-day long traces (compared to 7 days with the prototype). Unless otherwise stated, the simulator uses the same configuration as the prototype.

Baselines. We compare C2DN to three baselines. **(1) No-replication** does not provide fault tolerance and incurs no space overhead. **(2) Replication (CDN)** replicates each object with two replicas. We use the (CDN) suffix as this is most similar to the approach deployed today. **(3) C2DN-NoRebal** a C2DN variant based on consistent hashing without parity rebalance. In addition to C2DN, which uses one parity chunk and tolerates one unavailability, we have also evaluated C2DN-n5k3 and C2DN-n6k3, which uses two and three parity chunks, and can tolerate two and three unavailabilities, respectively.

6.2 Miss ratio without unavailability

We evaluate miss ratios of the competing systems under normal operation, i.e., *without unavailability*. Table 2 shows the object miss ratio and byte miss ratio of Replication (CDN) and C2DN obtained from the prototype experiments. We observe that C2DN reduces object miss ratio by 6.4% and byte miss ratio by 11.0%. These improvements are direct results of the reduced storage overhead in C2DN. At a large scale, these improvements lead to significant bandwidth savings.
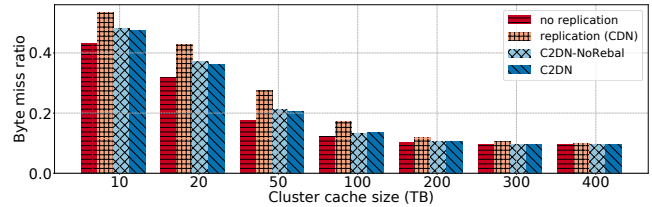

Figure 5: Byte miss ratio of the four systems.

To understand the sensitivity of byte miss ratio improvements to cache size, we show simulation results in Fig. 5. For smaller cache sizes, C2DN improves byte miss ratios by up to 20%. Benefits diminish for cache sizes above 200 TB ($5\times$ production cache size). For object miss ratios, the effect is qualitatively similar (Fig. 12 in the appendix). Overall, the reduction in miss ratio bridges the efficiency gap between No-replication and Replication (CDN) and reduces the overhead of providing redundancy in CDN edge clusters.

We also observe that C2DN improves miss ratios compared to C2DN-NoRebal because C2DN balances the write loads (eviction rates) across servers and reduces the probability of partial hits. However, this effect is small, suggesting that most of C2DN's miss ratio reduction comes from reduced storage overhead. The advantage of C2DN over C2DN-NoRebal will become clear in the following section, where we find that C2DN-NoRebal does not provide effective fault tolerance.

6.3 Miss ratio under unavailability

We now consider unavailabilities and evaluate the object miss ratio as the primary performance metric affecting latency and speedup. A first experiment introduces single unavailability after warming up the cache. We then measure the relative object miss ratio change: $\frac{mr(un)-mr(av)}{mr(av)}$ for each 5 minute time interval, where $mr(un)$ and $mr(av)$ stand for miss ratio with unavailability and without unavailability, respectively. A second experiment considers two simultaneous unavailabilities.

Fig. 6a show the relative object miss ratio increase where the single unavailability event occurs 100 minutes after warmup. As expected, No-replication does not provide fault tolerance, leading to a large ($2.2\times$ as seen in 1d) miss ratio spike. Replication (CDN) and C2DN-NoRebal have similar performance with 25% miss ratio spikes.

The miss ratio of C2DN is not affected for several hours after the unavailability event. This is because C2DN with one parity chunk can tolerate one unavailability effectively. In the long term, miss ratios for all systems increase as the cluster's total capacity is reduced. For C2DN, the increase in the miss ratio becomes visible only after around 300 minutes past unavailability. During unavailability, data that should be written to the unavailable servers are written to the other available servers. The extra writes take a long time to impact the miss ratio of clusters with a large cache size. We remark that the exact length of such no performance degradation is not fixed and is dependent on the trace.

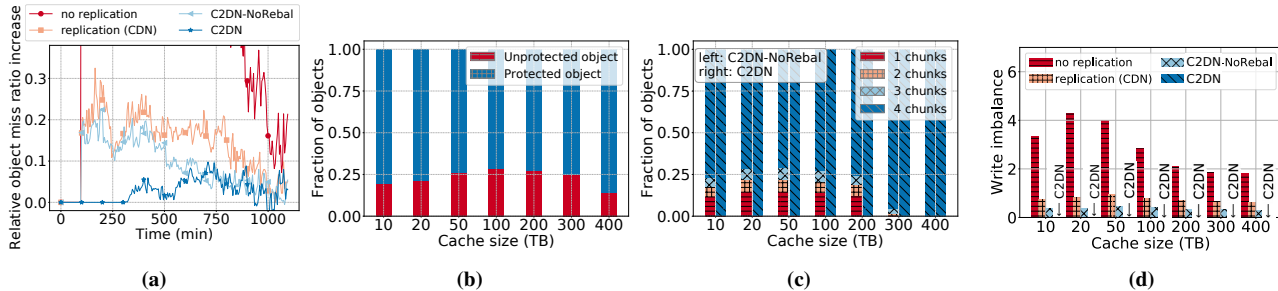The reason for the miss ratio spike in Replication (CDN)

Figure 6: a) Replicated CDN mitigates unavailability, but still has a spike after unavailability. C2DN mitigates the unavailability spike. b) Servers in the Replicated CDN evict objects independently, and due to write imbalance this leads to unprotected objects. c) Naive coding has similar problems as replication; C2DN solves this problem by parity rebalance. d) Write load imbalance for different systems across various cache sizes.
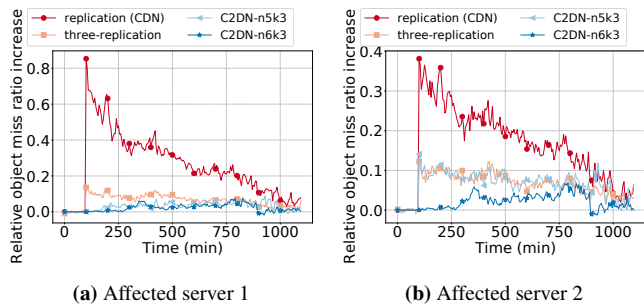

(a) Affected server 1   **(b)** Affected server 2

Figure 7: With two simultaneous unavailabilities, two replication (CDN) shows a big spike when the unavailability happens. Three replication and C2DN-n5k3 still show a small spike due to evicted replica/chunk. C2DN-n6k3 completely eliminates the spike.

System/server load	Max	Min	Mean	Max/min
CDN write (TB)	16.83	9.26	13.48	1.82
C2DN write (TB)	8.44	8.40	8.42	1.00

Table 3: Write load on servers in Replication (CDN) and C2DN.

(three) parity chunks with 66% (100%) storage overhead and can tolerate two (three) unavailabilities. In contrast, two-replication and three-replication tolerate one and two unavailabilities with 100% and 200% storage overhead, respectively.

Fig. 7 shows that compared to two-replication, C2DN-n5k3 and three-replication significantly reduce the miss ratio spike from over 80% to less than 20%. Furthermore, the miss ratio spike disappears entirely with C2DN-n6k3, which has the same storage overhead as two-replication.

6.4 Write (Read) load balancing

We quantify how well systems balance write load across servers. Balancing writes is the key to mitigating miss ratio spikes and helps control SSD tail latency and endurance.

Table. 3 shows bytes written per server in our prototype experiments. The busiest server in Replication (CDN) writes 16.8 TB compared to 8.4 TB for the busiest server in C2DN. With half the write rate, C2DN may double SSD lifetime and reduce tail latency by up to an order of magnitude [80]. The write imbalance in Replication (CDN) between peak and minimum write rate is $1.82\times$. In contrast, the write imbalance in C2DN is less than $1.005\times$. We also observe that C2DN reduces read imbalance from $1.69\times$ for Replication (CDN) to $1.34\times$. The read imbalance in C2DN remains as parity rebalancing (§4.2) focuses exclusively on write rate.

We further explore the effects of load balancing across various cache sizes using simulations. If M is the write (read) load on the server with maximum write (read) load and m is the minimum write (read) load across the servers, then write (read) load imbalance $= \frac{M-m}{m}$. Fig. 6d shows that C2DN eliminates write imbalance for all cache sizes. When averaged across cache sizes, C2DN reduces the **write** load imbalance by 99.9% compared to No-replication, 99.8% compared to Replication (CDN), and 99.5% compared to C2DN-NoRebal. C2DN also reduces the **read** load imbalance: by
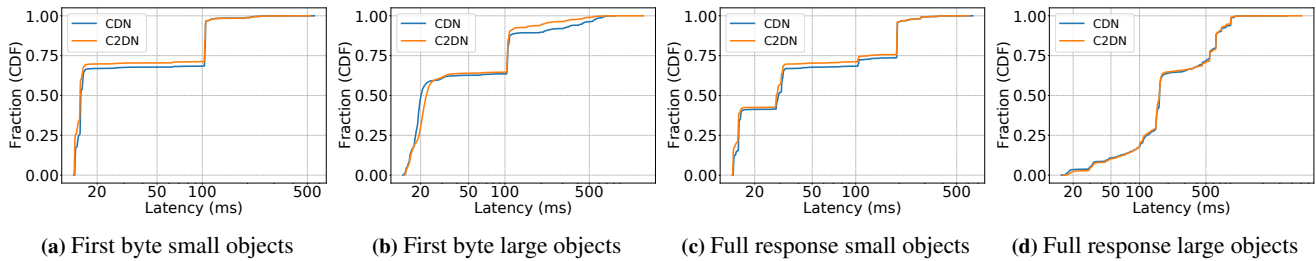
and C2DN-NoRebal— despite using redundancy — is the severe write and eviction rate imbalance in these systems (§3.3 and 3.4). This imbalance leads to *unprotected* objects: an object is unprotected if only a single copy is cached in the cluster. For C2DN-NoRebal, unprotected objects are objects with fewer than $K+1$ chunks cached in the cluster.

Fig. 6b shows the fraction of (un)protected objects in Replication (CDN). We observe that more than 25% of objects can be unprotected. The fraction of unprotected objects initially increases with cache size and then decreases. This pattern is because only highly popular objects are cached when the cache size is small, and hence the chance of having both replicas is higher. On the other hand, replicas are less likely to be evicted when the cache size is very large. Fig. 6c shows the fraction of unprotected objects in C2DN-NoRebal and C2DN. Since K =3 and P =1, objects with fewer than 4 chunks are unprotected. For caches smaller than 300 TB, up to 24% of the objects in C2DN-NoRebal are unprotected. In contrast, C2DN protects nearly 100% of cached objects across all cache sizes and effectively eliminates miss ratio spikes.

So far, we have only focused on one unavailability. When a CDN operator seeks to tolerate more than one unavailability, C2DN's advantage over replication increases as the space requirements for erasure coding scale significantly better. As an empirical data point, we consider two unavailabilities and compare two-replication and three-replication with C2DN-n5k3 and C2DN-n6k3. C2DN-n5k3 (C2DN-n6k3) uses two

(a) First byte small objects   **(b)** First byte large objects   **(c)** Full response small objects   **(d)** Full response large objects

Figure 8: First-byte and full response latency of serving small and large objects in CDN and C2DN.

93.9% compared to No-replication, 78.9% compared to Replication (CDN), and 70.5% compared to C2DN-NoRebal on an average across the different cache sizes.

6.5 Latency

We quantify potential latency overheads by measuring the time-to-first-byte (TTFB) and content download time (CDT) of our prototype implementations of C2DN and Replication (CDN). In each case, we separately measure the latency distribution for objects below the 128KB coding threshold ("small" objects) and for objects above the threshold ("large" objects). Fig. 8a and Fig. 8b show the cumulative distributions of TTFB for small and large objects, respectively. For small objects, we find that the TTFB distributions for C2DN and Replication (CDN) are similar, as expected: C2DN does not code these objects. C2DN slightly improves the TTFB distribution (shifting to the left) due to its lower object miss ratio. For large objects, we find about a 1 ms overhead in TTFB at low percentiles (25th-60th percentile). The slight increase is for cache hits due to fetching the first sub-chunk from K servers before serving the object.

We now consider CDT. In practice, this metric is more relevant for large objects than the TTFB. Fig. 8c and Fig. 8d show a cumulative distribution of the content download time for small and large objects, respectively. Again we find that small objects behave similarly in Replication (CDN) and C2DN, with slightly better latency for C2DN due to lower object miss ratio. For large objects, C2DN and Replication (CDN) have a similar CDT. The overheads of fetching chunks are hidden by our streaming implementation based on sub-chunks (§5).

We remark that C2DN improves the tail latency in all cases (barely visible in the CDFs). For example, C2DN reduces the P90 TTFB by up to 3× compared to Replication (CDN). We attribute this to a lower miss ratio and the mitigation of stragglers using parity chunks to serve requests. This is as expected based on prior work on using coding to reduce tail latency [55].

6.6 Overhead assessment

We quantify the resource overheads of our C2DN prototype.
CPU usage. Fig. 9a measures CPU utilization in fractional CPU cores for userspace and kernel tasks, respectively. C2DN generally leads to higher CPU usage. The userspace CPU usage is higher due to the encoding and decoding of objects, and
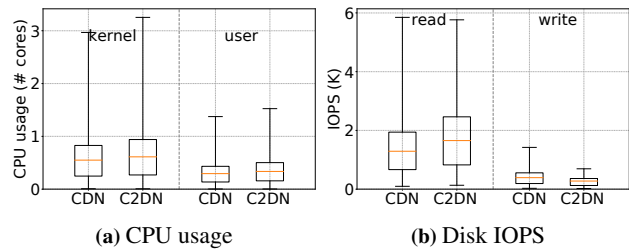

(a) CPU usage   **(b)** Disk IOPS

Figure 9: Resource usage. C2DN uses slightly more CPU resources and slightly more read disk IOPS than CDN, however, C2DN reduces write disk IOPS, especially at peak.

the kernel CPU usage is higher due to additional network and disk I/Os. Overall, CPU usage increases by 14% on average with a similar increase in kernel and userspace CPU usage.

The increase in the CPU overhead is small as C2DN performs the encoding and decoding operation only on a fraction of requests. For the current coding size threshold of 128 KB, the number of requests served with coded objects is around 50%, while the number of bytes served using coded objects is close to 90%. Also, recall that most requests for coded objects do not need to be decoded as the object is recreated by concatenating data chunks in the output buffer. Decoding only happens in the case of stragglers and partial hits. In fact, only 6% of requests require decoding in our experiments. These cases happen primarily due to the straggler problem (individual slow servers); actual data chunk misses (partial hits) occur for less than 0.6% of requests. A future version of C2DN may further reduce CPU overheads by using kernel-bypass networking or increasing the object size threshold for coding. Increasing the threshold can happen with minimal side effects, as we show in the next section.

Disk usage. Fig. 9b compares disk IOPS of Replication (CDN) and C2DN for reads and writes. For reads, we observe that C2DN uses 23% more IOPS in the mean and less at the tail. Read-IOPS increase by 2% at the P99 and decrease by 11% at the P99.9 (we calculate this percentile across time and servers). For writes, C2DN uses 24% fewer writes IOPS in the mean. The tail write IOPS decreases by 46% at the P99 and 50% at the P99.9. The read IOPS increases because C2DN fetches at least $K = 3$ chunks to serve an object if coded. However, due to 1) most of the requests being for small uncoded objects, 2) the presence of DRAM hot object cache, the increase in reading IOPS is much smaller than 3×. While mean read IOPS increase, the peak read IOPS is similar or lower in C2DN. We attribute this to better load balancing

in C2DN. Write IOPS in C2DN is significantly reduced when compared to Replication (CDN). C2DN has a lower storage overhead than Replication (CDN) and thus writes less to disk. In addition, the improvement in the miss ratio that C2DN provides further reduces the number of write operations. Besides, C2DN also improves the tail write IOPS, which is due to a better load balancing strategy of erasure coding and parity rebalance.

Intra-cluster network usage. C2DN uses network bandwidth within the cluster, about 0.9 Gbps in the mean and 2.3 Gbps at the P95. In conversations with CDN operators, this internal bandwidth usage is feasible for production clusters, as these links generally show little usage. For example, production CDN clusters use dedicated 10-Gbps-NICs for communication within the cluster.

6.7   Sensitivity analysis

We discuss the sensitivity of C2DN to its parameters.

Coding size threshold. The size threshold for coding impacts the performance in multiple ways. By reducing the size threshold, C2DN encodes more objects, improving cache space usage and load balance across cluster servers. At the same time, it leads to more CPU and I/Os (due to coding and fetching) and increases the latency for small objects. The size distribution in Fig. 1a shows that small objects contribute a small fraction of cache space usage. Thus, the potential benefit of coding diminishes as we decrease the size threshold for coding. At the same time, C2DN would use more cluster resources. We observe that reducing the size threshold to below 128 KB does not significantly benefit the object and byte miss ratio. Increasing the size threshold to over 8 MB increases the byte miss ratio by 2.79% and the write load imbalance by 258%. We believe that 128 KB is a good tradeoff for our production traces. Fig. 13 in the appendix shows our results.

Coding parameter K. Most of this section assumed C2DN configured with $K = 3$. We explore the impact of parameter K and P on miss ratio and write load balancing. We find that increasing K and keeping P constant reduces miss ratios for C2DN but increases miss ratios for C2DN-NoRebal. When adding chunks, the probability of getting partial hits increases for C2DN-NoRebal due to unbalanced eviction rates between servers. Because C2DN uses parity rebalance to achieve similar eviction rates between servers, the miss ratio decreases with increasing K due to lower storage overhead. While the impact of coding parameters has different impacts on miss ratios for C2DN-NoRebal and C2DN, the impact on load balancing is similar, as K increases, because an object is broken into more (and smaller) chunks, both the read and write load imbalance in C2DN-NoRebal and C2DN reduce. Fig. 14 in the appendix shows our results.

Different workloads. Throughout this section, we have used the video trace. We repeated our evaluation for the week-long web trace (§3). Compared to the video trace, the web trace has a significantly smaller working set. The video trace has a

compulsory byte miss ratio of 0.1 and a compulsory object miss ratio of 0.21. In the web trace, the compulsory miss ratio is 0.06 for both byte and object miss ratios. In addition, compared to the video trace, the web trace has a more diverse object size range, as shown in Fig. 1a. Less than 10% of large objects (possibly large software) contribute to more than 90% of the cache space usage. Therefore, the fraction of requests that require coding is significantly smaller.

In prototype experiments with the web trace, only 3% of all requests are served coded. However, these 3% of requests account for 80% of served traffic. As a comparison, in the video trace, the prototype serves about 50% of requests from coded objects (with only 6% requiring decoding). Consequently, coding overheads on the web trace are negligible. In terms of the miss ratio, we observe a 10% reduction in object miss ratio and a 6% reduction in byte miss ratio. The write imbalance for Replication (CDN) is $1.72\times$, which is reduced to $1.03\times$ in C2DN. The read imbalance for Replication (CDN) is $4.8\times$, which is reduced to $2.5\times$ in C2DN.

Different eviction algorithms. Throughout this section, we have used FIFO as the eviction algorithm for the cache. FIFO provides stable performance on SSDs and extends the lifetime of an SSD by minimizing device write amplification [9,22,70]. Many open source caches such as Apache Trafficserver [7] and Varnish [71] use FIFO. To understand the impact of the eviction algorithm, we evaluate the Least-recently-used algorithm (LRU) using simulation. We observe a slight reduction in both object and byte miss ratios for all systems. All other results are qualitatively and quantitatively the same. Appendix 10.4, Fig. 16 and Fig. 17 show these results.

Variants of replication. Besides two-replication for all objects, CDNs have explored systems that replicate based on popularity. Specifically, only popular objects are replicated on two servers to reduce space overheads. As might be expected from our findings that write imbalance matters, popularity-based replication does not provide good fault tolerance. In simulation experiments, we observe object miss ratio spikes by 82%. Interestingly, we also observe that popularity-based replication leads to an even worse load imbalance than Replication (CDN), which explains the high miss ratio spike.

7   Discussion

DNS vs anycast-based CDN request routing. Different CDNs use different global load balancing architectures [33]. Akamai is well known for its DNS architecture [64]. Limelight [33], Wikipedia [58], and Cloudflare rely on anycast. While these designs have different performance implications, both rely on algorithms like consistent hashing. In DNS-based systems, consistent hashing is applied by the cluster-local load balancer to return the IP of the server responsible for the shard. Anycast-based systems typically route requests to any server in a cluster, and the server uses consistent hashing to identify another server that likely stores the object. Server

unavailability, storage overheads of redundancy, and write imbalance are important problems in all CDN designs. While the cluster-local load balancer in our prototype relies on DNS, the principle design components of C2DN can be equally applied in anycast systems. We also expect that C2DN's benefits will transfer with similar quantitative improvements.

Larger clusters and multiple unavailabilities. In clusters of large size, multiple concurrent unavailabilities are not uncommon. As evaluated in §6, we find that C2DN is more effective in this setting as erasure coding is more efficient at tolerating multiple unavailabilities than replication. For large clusters, server unavailabilities become more common. We thus recommend either using a coding scheme with more parity chunks or handling the cluster as multiple smaller clusters.

8 Related work

While there is extensive work on caching, coding, load balancing, and flash caching, our work is uniquely positioned at the intersection of these areas. We discuss work by area.

Erasure coding in storage systems. Prior work has characterized the cost advantage offered by coding over replication in achieving data durability in distributed storage systems [75, 85]. Erasure codes are deployed in RAID [52], network-attached-storage [4], peer-to-peer storage systems [37, 40, 57, 79], in-memory key-value store [16, 17, 84], and distributed storage systems [32, 48, 56, 76]. Coding for CDNs differs due to the unique interplay of coding and caching and the two-sided transparency requirement (§4). Additionally, CDNs employ coding for different reasons (performance) than storage systems (durability), which magnifies overhead concerns.

Caching for coded file systems. Several recent works have explored augmenting erasure-coded storage systems with a cache to reduce latency [3, 29, 41, 55]. Aggarwal et al. [3] proposed augmenting erasure-coded disk-based storage systems with an in-memory cache at the proxy or the client-side that cache encoded chunks. Halalai et al. [29] propose augmenting geo-distributed erasure-coded storage systems by caching a fraction of the coded chunks in different geo-locations to alleviate the latency impact of fetching chunks from remote geo-locations. EC-Cache [55] employs erasure coding in the in-memory layer of a tiered distributed file system such as Alluxio (formerly [39]). Although EC-Cache is technically a cache, there is no interaction between coding and caching in EC-Cache since it operates in scenarios where the entire working set fits in memory, i.e., no evictions are considered. In contrast, C2DN focuses on CDN clusters with working sets in the hundreds of TB and starkly different tradeoffs, workload characteristics, and challenges as compared to file systems. In the area of cooperative caching [6, 30, 62], nodes synchronize caching decision via explicit communication. In contrast, C2DN proves that explicit communication is not required to synchronize the eviction of the K chunks, which significantly decreases overheads.

Chunking and caching. Prior work has explored the challenge of serving large files over HTTP, e.g., CoDeeN [74]. Similar to C2DN, CoDeeN breaks a large file into smaller chunks. A chunk cache miss does not require transferring the whole large file from the origin. In contrast to CodeeN, C2DN addresses unavailability tolerance, which is not provided by chunking alone.

Load balancing. Load balancing and sharding are well-studied topics [1, 2, 12, 13, 21, 27, 28]. To reduce the load imbalance, John et al. study the power of two choices that reduces the imbalance [12]. In addition, to serve skewed workloads, Fan et al. [24] study the effect of using a small and fast popularity-based cache to reduce load imbalance between different caches in a large backend pool. Yu-ju et al. [31] designed SPORE to use a self-adapting, popularity-based replication to mitigate load imbalance. Rashmi et al. [55] used erasure coding to reduce read load imbalance for large object in-memory cache. In summary, prior work on load balancing focuses on *read* load balancing, with little attention paid to *write* load balancing.

Load imbalance in consistent hashing can be solved with additional lookups via probing [47]. Unfortunately, these lookups are costly in CDNs (particularly for DNS-based systems). Additionally, this approach cannot be applied for erasure-coded caches due to the constraint that parity chunks should not be colocated with data chunks. In contrast to using load balancing to achieve a similar SSD replacement time, Mahesh et al. [8] used parity placement to achieve differential SSD ages so that SSDs of a disk array fail at different times.

Flash cache endurance. Flash caching is an active and challenging research area. A line of work [38, 45, 51, 63, 66, 67, 70] shows how eviction policies can be efficiently implemented on flash. Flashield [22] proposes to extend SSD lifetime via smart admission policies. All these systems focus on a single SSD. Our work focuses on wear-leveling across servers in a cluster, which significantly extends the lifetime of a cluster.

9 Conclusion

We re-architected the cluster of a CDN by introducing a hybrid redundancy scheme using erasure codes and replication, along with a novel approach for parity placement. We showed that our approach reduces the miss ratio and eliminates the miss ratio spikes caused by server unavailability. Further, our approach is more space-efficient than replication and is more attractive as CDN traffic and content footprint scale rapidly with Internet usage. Finally, our approach also reduces the write load imbalance by optimally placing the parities to reduce the lifetime of SSDs. We believe that C2DN is attractive for deployment in a production CDN since it integrates well with production CDN components.

References

[1] A. Adya, J. Dunagan, and A. Wolman. Centrifuge: Integrated lease management and partitioning for cloud services. In *7th USENIX Symposium on Networked Systems Design and Implementation (NSDI 10)*, San Jose, CA, Apr. 2010. USENIX Association.

[2] A. Adya, D. Myers, J. Howell, J. Elson, C. Meek, V. Khemani, S. Fulger, P. Gu, L. Bhuvanagiri, J. Hunter, et al. Slicer: Auto-sharding for datacenter applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 739–753, 2016.

[3] V. Aggarwal, Y.-F. R. Chen, T. Lan, and Y. Xiang. Sprout: A functional caching approach to minimize service latency in erasure-coded storage. In *ICDCS*, 2016.

[4] M. Aguilera, R. Janakiraman, and L. Xu. Using erasure codes efficiently for storage in a distributed system. In *DSN*, 2005.

[5] A. V. Aho, P. J. Denning, and J. D. Ullman. Principles of optimal page replacement. *Journal of the ACM (JACM)*, 18(1):80–93, 1971.

[6] S. Annapureddy, M. J. Freedman, and D. Mazieres. Shark: Scaling file servers via cooperative caching. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 129–142. USENIX Association, 2005.

[7] Apache. Traffic Server, 2019. Available at https://trafficserver.apache.org/, accessed 09/18/19.

[8] M. Balakrishnan, A. Kadav, V. Prabhakaran, and D. Malkhi. Differential raid: Rethinking raid for ssd reliability. *ACM Transactions on Storage (TOS)*, 6(2):1–22, 2010.

[9] B. Berg, D. S. Berger, S. McAllister, I. Grosof, S. Gunasekar, J. Lu, M. Uhlar, J. Carrig, N. Beckmann, M. Harchol-Balter, et al. The cachelib caching engine: Design and experiences at scale. In *USENIX OSDI*, pages 753–768, 2020.

[10] D. S. Berger, R. Sitaraman, and M. Harchol-Balter. Adaptsize: Orchestrating the hot object memory cache in a cdn. In *USENIX NSDI*, pages 483–498, March 2017.

[11] M. Bjørling, J. Gonzalez, and P. Bonnet. Lightnvm: The linux open-channel SSD subsystem. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 359–374, Santa Clara, CA, Feb. 2017. USENIX Association.

[12] J. Byers, J. Considine, and M. Mitzenmacher. Simple load balancing for distributed hash tables. In *International Workshop on Peer-to-Peer Systems*, pages 80–87. Springer, 2003.

[13] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin. Orleans: Cloud computing for everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, New York, NY, USA, 2011. Association for Computing Machinery.

[14] D. G. Cattrysse and L. N. Van Wassenhove. A survey of algorithms for the generalized assignment problem. *European journal of operational research*, 60(3):260–272, 1992.

[15] F. Chen, R. K. Sitaraman, and M. Torres. End-user mapping: Next generation request routing for content delivery. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 167–181. ACM, 2015.

[16] H. Chen, H. Zhang, M. Dong, Z. Wang, Y. Xia, H. Guan, and B. Zang. Efficient and available in-memory kv-store with hybrid erasure coding and replication. *ACM Transactions on Storage (TOS)*, 13(3):1–30, 2017.

[17] L. Cheng, Y. Hu, and P. P. C. Lee. Coupling decentralized key-value stores with erasure coding. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 377–389, New York, NY, USA, 2019. Association for Computing Machinery.

[18] B. V. Cherkassky and A. V. Goldberg. On implementing the push—relabel method for the maximum flow problem. *Algorithmica*, 19(4):390–410, 1997.

[19] CISCO. Global IP traffic forecast: The zettabyte era—trends and analysis, June 2017. Available at https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/vni-hyperconnectivity-wp.pdf, accessed 24/09/17.

[20] J. Dilley, B. M. Maggs, J. Parikh, H. Prokop, R. K. Sitaraman, and W. E. Weihl. Globally distributed content delivery. *IEEE Internet Computing*, 6(5):50–58, 2002.

[21] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein. Maglev: A fast and reliable software network load balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 523–535, 2016.

[22] A. Eisenman, A. Cidon, E. Pergament, O. Haimovich, R. Stutsman, M. Alizadeh, and S. Katti. Flashield: a hybrid key-value cache that controls flash write amplification. In *16th USENIX Symposium on Networked*

Systems Design and Implementation (NSDI 19), pages 65–78, Boston, MA, Feb. 2019. USENIX Association.

[23] R. Fagin. Asymptotic miss ratios over independent references. *Journal of Computer and System Sciences*, 14(2):222–250, 1977.

[24] B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky. Small cache, big effect: Provable load balancing for randomly partitioned cluster services. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, pages 1–12, 2011.

[25] D. Ford, F. Labelle, F. Popovici, M. Stokely, V. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in globally distributed storage systems. In *USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[26] C. Fricker, P. Robert, and J. Roberts. A versatile and accurate approximation for LRU cache performance. In *ITC*, page 8, 2012.

[27] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang. Duet: Cloud scale load balancing with hardware and software. *ACM SIGCOMM Computer Communication Review*, 44(4):27–38, 2014.

[28] A. Gulati, C. Kumar, I. Ahmad, and K. Kumar. Basil: Automated io load balancing across storage devices. In *Fast*, volume 10, pages 13–13, 2010.

[29] R. Halalai, P. Felber, A.-M. Kermarrec, and F. Taïani. Agar: A caching system for erasure-coded data. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 23–33. IEEE, 2017.

[30] V. Holmedahl, B. Smith, and T. Yang. Cooperative caching of dynamic content on a distributed web server. In *Proceedings. The Seventh International Symposium on High Performance Distributed Computing (Cat. No. 98TB100244)*, pages 243–250. IEEE, 1998.

[31] Y.-J. Hong and M. Thottethodi. Understanding and mitigating the impact of load imbalance in the memory caching tier. In *Proceedings of the 4th annual Symposium on Cloud Computing*, pages 1–17, 2013.

[32] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure coding in Windows Azure Storage. In *USENIX ATC*, 2012.

[33] C. Huang, A. Wang, J. Li, and K. W. Ross. Measuring and evaluating large-scale cdns. In *ACM IMC*, volume 8, pages 15–29, 2008.

[34] B. Jiang, P. Nain, and D. Towsley. On the convergence of the ttl approximation for an lru cache under independent stationary request processes. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 3(4), Sept. 2018.

[35] S. Kadekodi, F. Maturana, S. J. Subramanya, J. Yang, K. Rashmi, and G. R. Ganger. {PACEMAKER}: Avoiding heart attacks in storage clusters with disk-adaptive redundancy. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 369–385, 2020.

[36] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM Symposium on Theory of Computing*, pages 654–663, 1997.

[37] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, et al. Oceanstore: An architecture for global-scale persistent storage. *ACM Sigplan Notices*, 35(11):190–201, 2000.

[38] C. Li, P. Shilane, F. Douglis, and G. Wallace. Pannier: Design and analysis of a container-based flash cache for compound objects. *ACM Transactions on Storage (TOS)*, 13(3):1–34, 2017.

[39] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *SoCC*, 2014.

[40] J. Li and C. Zhang. Distributed hosting of web content with erasure coding and unequal weight assignment. In *2004 IEEE International Conference on Multimedia and Expo (ICME) (IEEE Cat. No.04TH8763)*, volume 3, pages 2087–2090 Vol.3, 2004.

[41] K. Liu, J. Peng, J. Wang, and J. Pan. Optimal caching for low latency in distributed coded storage systems. *arXiv preprint arXiv:2012.03005*, 2020.

[42] R. S. Liu, C. L. Yang, C. H. Li, and G. Y. Chen. Duracache: A durable ssd cache using mlc nand flash. In *Proceedings of the 50th Annual Design Automation Conference*, pages 1–6, 2013.

[43] B. M. Maggs and R. K. Sitaraman. Algorithmic nuggets in content delivery. *SIGCOMM Comput. Commun. Rev.*, 45(3):52–66, July 2015.

[44] V. Martina, M. Garetto, and E. Leonardi. A unified approach to the performance analysis of caching systems. In *IEEE INFOCOM*, 2014.

[45] S. McAllister, B. Berg, J. Tutuncu-Macias, J. Yang, S. Gunasekar, J. Lu, D. S. Berger, N. Beckmann, and G. R. Ganger. Kangaroo: Caching billions of tiny objects on flash. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 243–262, New York, NY, USA, 2021. Association for Computing Machinery.

[46] J. Meza, T. Xu, K. Veeraraghavan, and O. Mutlu. A large scale study of data center network reliability. In *Proceedings of the Internet Measurement Conference 2018*, pages 393–407, 2018.

[47] V. Mirrokni, M. Thorup, and M. Zadimoghaddam. Consistent hashing with bounded loads. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 587–604, 2018.

[48] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, et al. f4: Facebook's warm BLOB storage system. In *USENIX OSDI*, pages 383–398, 2014.

[49] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, et al. Scaling memcache at Facebook. In *USENIX NSDI*, pages 385–398, 2013.

[50] E. Nygren, R. K. Sitaraman, and J. Sun. The Akamai Network: A platform for high-performance Internet applications. *ACM SIGOPS Operating Systems Review*, 44(3):2–19, 2010.

[51] Y. Oh, J. Choi, D. Lee, and S. H. Noh. Caching less for better performance: balancing cache size and update cost of flash memory cache in hybrid storage systems. In *FAST*, volume 12, 2012.

[52] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *SIGMOD*, 1988.

[53] L. Perron and V. Furnon. OR-tools. https://developers.google.com/optimization/.

[54] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A hitchhiker's guide to fast and efficient data reconstruction in erasure-coded data centers. In *SIGCOMM*, 2015.

[55] K. V. Rashmi, M. Chowdhury, J. Kosaian, I. Stoica, and K. Ramchandran. Ec-cache: Load-balanced, low-latency cluster caching with online erasure coding. In *USENIX OSDI*, pages 401–417, 2016.

[56] K. V. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the facebook warehouse cluster. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Storage and File Systems*, 2013.

[57] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: The OceanStore prototype. In *FAST*, 2003.

[58] E. Rocca. Running Wikipedia.org, June 2016. available https://www.mediawiki.org/wiki/File:WMF_Traffic_Varnishcon_2016.pdf accessed 09/12/16.

[59] R. K. Sahoo, M. S. Squillante, A. Sivasubramaniam, and Y. Zhang. Failure data analysis of a large-scale heterogeneous server environment. In *International Conference on Dependable Systems and Networks, 2004*, pages 772–781, 2004.

[60] L. Saino, I. Psaras, and G. Pavlou. Understanding sharded caching systems. In *IEEE INFOCOM*, pages 1–9, 2016.

[61] Sanjay Sane. Latency and wear-out in facebook's cdn due to ssd write pressure. Private conversation,, 7 2019.

[62] P. Sarkar and J. H. Hartman. Hint-based cooperative caching. *ACM Transactions on Computer Systems (TOCS)*, 18(4):387–419, 2000.

[63] M. Saxena, M. M. Swift, and Y. Zhang. Flashtier: A lightweight, consistent and durable storage cache. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, page 267–280, New York, NY, USA, 2012. Association for Computing Machinery.

[64] K. Schomp, O. Bhardwaj, E. Kurdoglu, M. Muhaimen, and R. K. Sitaraman. Akamai DNS: Providing authoritative answers to the world's queries. In *ACM SIGCOMM*, pages 465–478, 2020.

[65] B. Schroeder and G. Gibson. A large-scale study of failures in high-performance computing systems. *IEEE transactions on Dependable and Secure Computing*, 7(4):337–350, 2009.

[66] Z. Shen, F. Chen, Y. Jia, and Z. Shao. Optimizing flash-based key-value cache systems. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, Denver, CO, June 2016. USENIX Association.

[67] Z. Shen, F. Chen, Y. Jia, and Z. Shao. Didacache: an integration of device and application for flash-based key-value caching. *ACM Transactions on Storage (TOS)*, 14(3):1–32, 2018.

[68] A. Sundarrajan, M. Feng, M. Kasbekar, and R. K. Sitaraman. Footprint descriptors: Theory and practice of cache provisioning in a global cdn. In *ACM CoNEXT*, pages 55–67, 2017.

[69] A. Sundarrajan, M. Kasbekar, R. K. Sitaraman, and S. Shukla. Midgress-aware traffic provisioning for content delivery. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 543–557, 2020.

[70] L. Tang, Q. Huang, W. Lloyd, S. Kumar, and K. Li. Ripq: Advanced photo caching on flash for facebook. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 373–386, 2015.

[71] F. Velázquez, K. Lyngstøl, T. Fog Heen, and J. Renard. *The Varnish Book for Varnish 4.0*. Varnish Software AS, March 2016.

[72] C. Waldspurger, T. Saemundsson, I. Ahmad, and N. Park. Cache modeling and optimization using miniature simulations. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 487–498, 2017.

[73] C. A. Waldspurger, N. Park, A. Garthwaite, and I. Ahmad. Efficient mrc construction with shards. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 95–110, 2015.

[74] L. Wang, K. Park, R. Pang, V. S. Pai, and L. L. Peterson. Reliability and security in the codeen content distribution network. In *USENIX ATC*, pages 171–184, 2004.

[75] H. Weatherspoon and J. D. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *IPTPS*, 2002.

[76] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *OSDI*, 2006.

[77] G. Wu and X. He. Reducing ssd read latency via nand flash program and erase suspension. In *FAST*, volume 12, pages 10–10, 2012.

[78] K. Wu, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Towards an unwritten contract of intel optane SSD. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, Renton, WA, July 2019. USENIX Association.

[79] F. Xu, Y. Wang, and X. Ma. Online encoding for erasure-coded distributed storage systems. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 338–342, 2017.

[80] S. Yan, H. Li, M. Hao, M. H. Tong, S. Sundararaman, A. A. Chien, and H. S. Gunawi. Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in nand ssds. *ACM Trans. Storage*, 13(3), Oct. 2017.

[81] J. Yang, Y. Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 191–208. USENIX Association, Nov. 2020.

[82] J. Yang, Y. Yue, and K. V. Rashmi. A large-scale analysis of hundreds of in-memory key-value cache clusters at twitter. *ACM Trans. Storage*, 17(3), aug 2021.

[83] J. Yang, Y. Yue, and R. Vinayak. Segcache: a memory-efficient and scalable in-memory key-value cache for small objects. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 503–518. USENIX Association, Apr. 2021.

[84] M. M. T. Yiu, H. H. W. Chan, and P. P. C. Lee. Erasure coding for small objects in in-memory kv storage. In *Proceedings of the 10th ACM International Systems and Storage Conference*, SYSTOR '17, New York, NY, USA, 2017. Association for Computing Machinery.

[85] Z. Zhang, A. Deshpande, X. Ma, E. Thereska, and D. Narayanan. Does erasure coding have a role to play in my data center? Technical Report Microsoft Research MSR-TR-2010, 2010.

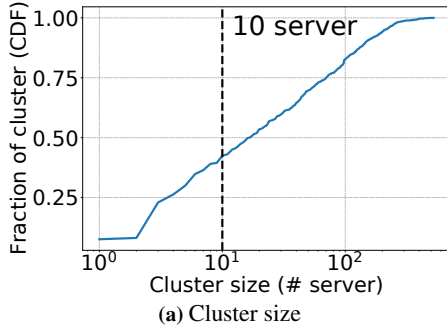# 10 Supplemental information

10.1 Cluster size distribution

(a) Cluster size

Figure 10: Cluster size ranges from 1 to over 500 servers.

10.2 Proof details

Proof of Theorem 1.
Let T_c^i denote the *characteristic time* [23, 26] of the cache at server i, given capacity C, where the characteristic time of an LRU cache measures how long it takes for a newly requested chunk to get evicted. We first prove that for any two servers i and j, T_c^i and T_c^j are nearly the same. More precisely, we show that for any $i \neq j$, $Prob(|T_c^i - T_c^j| \geq \varepsilon)$ is at most $O(W/(\varepsilon^2 C))$, using a mathematical argument similar to [60]. Where C denotes the cache size of each server and W is the variance of the write load imbalance across the servers in the cluster. Due to parity rebalancing in C2DN, $W \to 0$. So, this probability $O(W/(\varepsilon^2 C))$ vanishes as the cache size grows large.

In C2DN, when an object is requested, its chunks $x_1, \dots, x_n$ are requested at the same time from the individual servers. Since the characteristic time of the servers that these chunks reside in are (nearly) the same as shown above, it follows that these chunks are evicted from these caches at (nearly) the same time. Thus, the chunks $x_1, \dots, x_n$ of an object enter and exit their individual caches in a synchronized way, even though there is no explicit coordination among the caches.

10.3 Additional details on parity rebalance

Here, we give more details about the bucket assignment algorithm discussed in §4 under the three scenarios (1) Initial bucket assignment, (2) Server failure, (3) Server addition. We first consider the case where the servers are homogeneous and later extend the algorithm to the heterogeneous case.
Initial bucket assignment. The algorithm runs in two phases. In the first phase, the data buckets are assigned to the servers using the consistent hashing algorithm. The algorithm chooses K consecutive servers on the consistent hash ring

from the bucket's hash location in a clockwise direction to assign the K data chunk buckets. In the second phase, the parity chunks are assigned to the servers such that the load is evenly balanced across the servers. The second phase is described in Algorithm 1.

Algorithm 1 Phase 2. Parity rebalance
1: **Input** : Set of available servers $\mathcal{A}$ and the total write load on the cluster W.
2: Set of $\mathcal{N}$ parity buckets. For $n \in \mathcal{N}$, the sum of sizes of the parity chunks in the bucket is s_n.
3: A set L with l_i denoting the current write load (due to assignment of data buckets and uncoded objects) on server i.
4: **Output** : A valid assignment of parity bucket to the servers.
5: **Initialize** : A set of vertices $V \leftarrow \phi$, a set of edges $E \leftarrow \phi$, an empty graph $\mathcal{G} = (V, E)$.
6: Add source node S, terminal node T, nodes corresponding to parity buckets and available servers to V.

7: **for** $n \in \mathcal{N}$ **do** // Loop through parity buckets
8:     $e \leftarrow ((S,n), n_s)$ // Create edge between nodes between source-node S and bucket-node n with weight equal to size of the bucket n_s
9:     $V \leftarrow n, E \leftarrow e$
10:     **for** $a \in \mathcal{A}$ **do**
11:         **if** data chunk of bucket n is not assigned to a **then**
12:             $e \leftarrow ((n,a), n_s)$ // Create edge between parity bucket node n and available server a with size of parity bucket n_s.
13:         **end if**
14:     **end for**
15: **end for**

16: **for** $a \in \mathcal{A}$ **do** // Loop through available servers
17:     $c_a \leftarrow max(\lceil \frac{W}{|\mathcal{A}|} \rceil - l_i, 0)$ // Available budget on each server
18:     $e \leftarrow ((a,T), c_a)$ // Create edge between server nodes a and sink-node T with weight c_a
19:     $V \leftarrow v, E \leftarrow e$
20: **end for**

21: Run MaxFlow(S, T) between the source-node S and terminal node T.
22: **for** Each parity bucket $n \in \mathcal{N}$ **do**
23:     Assign the parity bucket to the least loaded server that has a positive assigned flow from the bucket.
24: **end for**

Post running the Phase 1 of the algorithm, the available write budget on each server a (c_a) in line 17 of Algorithm 1, is obtained as follows. If the total unique bytes requested to the cluster is W, then each server should host traffic not more than $\lceil \frac{W}{|\mathcal{A}|} \rceil$ bytes. After phase 1 of the algorithm, if l_i is the traffic assigned to server i, then the available budget on server i i.e., c_i is obtained as $max(\lceil \frac{W}{|\mathcal{A}|} \rceil - l_i, 0)$. An empty graph is initialized in line 5 and the source and terminal nodes are added in line 6. In line 8, for each bucket n we add an edge from the source node S to the bucket-node with a capacity

that equals the size of the parity bucket. Through lines 10-14, for each bucket *n* add a directed edge from the node that corresponds to the parity bucket to a server-node that is not assigned a data chunk of bucket *n*. These edges capture if the parity bucket can be assigned to a server. Then we assign these edges with a capacity that equals to the size of the parity bucket. In lines 16-19 directed edges are added from server nodes (*a*) to the sink node with a capacity c_a i.e., the available write budget on the server. Now run the max-flow algorithm in the graph between the source-node *S* and the sink-node *T* to find a valid assignment of parity buckets to the servers. To assign a parity bucket to a server, we find edges from the parity-node to the server-nodes that are assigned a positive flow. The servers are potential candidates for assignment. Empirically, we find that in most runs, the algorithm finds a single candidate server, if not, we assign the parity bucket to the least loaded server among the potential candidates.

Server failure. When a server fails the data buckets (*D*) and parity buckets (*P*) belonging to the server need to be reassigned. As done previously, the data buckets are reassigned using the consistent hash ring. Now, the new data bucket allocation could invalidate some of the previous parity bucket assignments (on the currently available servers) as the data chunks and parity chunks cannot cohabit the same server. Let the invalidated parity buckets be P'. The available budget c_i of each server *i* is recalculated using the total traffic *W* and the number of available servers $|\mathcal{A}|$. Now, the buckets $P \cup P'$ (parity buckets of failed server and invalidated parity buckets) are assigned to the servers using Algorithm 1 by reassigning corresponding capacities in line 9. When a server fails we also keep track of the parity buckets that were assigned to it before failure. Some of these buckets could be reassigned to the server when it is available again depending on the available write budget at each server.

Server addition. When a server is available again, it gets assigned the data buckets using the consistent hashing algorithm. We recompute the capacity of each server as done previously. Now, as we have kept track of the parity buckets the server was assigned prior to failure, we try to re-assign as many of those parity buckets as possible. If we get back all buckets and we still have capacity for more buckets to be assigned, we iteratively pick parity buckets from the most loaded server.

Algorithm complexity. In Algorithm 1 the cost of constructing the graph can be computed using $O(|T| \times |\mathcal{N}|)$ time complexity where *T* is the number of servers. And the time complexity is because we need to decide if we wish to add an edge between the parity chunk and the server. Further, the runtime complexity of the MaxFlow algorithm in line 21 is computed as follows. The total available budget on the servers is $B = \sum_{i \in S} \left(\left\lceil \frac{W}{|\mathcal{A}|} \right\rceil - l_i \right)$. Then, the runtime complexity is $B \times |\mathcal{N}| \times |T|)$.

Heterogeneous servers. We extend consistent-hashing-based bucket assignment to the case of heterogeneous servers. Each
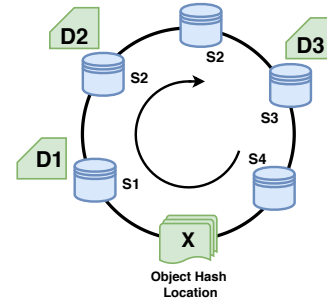

Figure 11: Consistent Hashing for a cluster with heterogeneous servers. The data buckets D1, D2 and D3 are hashed to unique servers S1, S2 and S3.

server is represented by at least one virtual node on the consistent hashing ring. The number of virtual nodes added is proportional to the capacity of the server (e.g., if server A is $2\times$ larger than server B, then server A would have $2\times$ as many virtual nodes).

The data buckets are mapped to the consistent hashing ring as follows. From the bucket's hashed position on the ring, we move along the ring in a clockwise direction and assign — the *K* data buckets — iteratively to the virtual nodes encountered. However, while assigning, we step over servers that have been assigned any of the other *K* data buckets. This ensures each of the K data buckets are assigned to different servers. Figure 11 shows a placement example. The cluster consists of 4 servers S1, S2, S3 and S4 of capacity C, 2C, C, C respectively. Virtual nodes are indicated by the server name. Note that, as S2 is twice the capacity of the servers, we create two virtual nodes for S2. By chance, we assume that the two virtual nodes hash to adjacent positions on the ring. Now, if the bucket is hashed to a location X on the consistent hash ring, then the data buckets D1, D2 and D3 are assigned to the first three servers encountered by moving in the clockwise direction from X. This is S1, S2 (S2 again and thus skipped), and S3, which is the resulting bucket placement. After data buckets are assigned, we use Algorithm 1 to assign parity buckets.
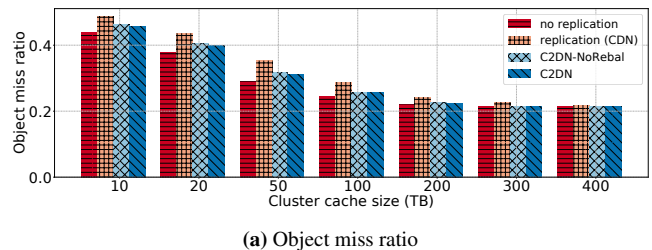

(a) Object miss ratio

Figure 12: Object miss ratio of different systems.

10.4 Additional figures for sensitivity analysis

(a) Object miss ratio

(b) Byte miss ratio

(c) Read load balancing

(d) Write load balancing
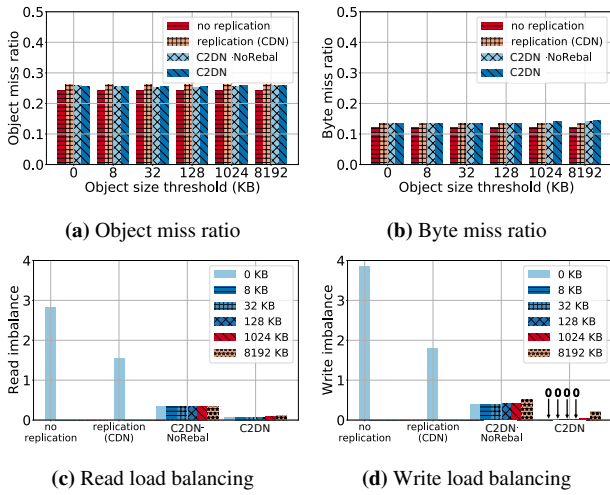
Figure 13: Impact of coding size threshold on miss ratio and load balancing.

(a) Impact on object miss ratio

(b) Impact on byte miss ratio

(c) Impact on read imbalance

(d) Impact on write imbalance
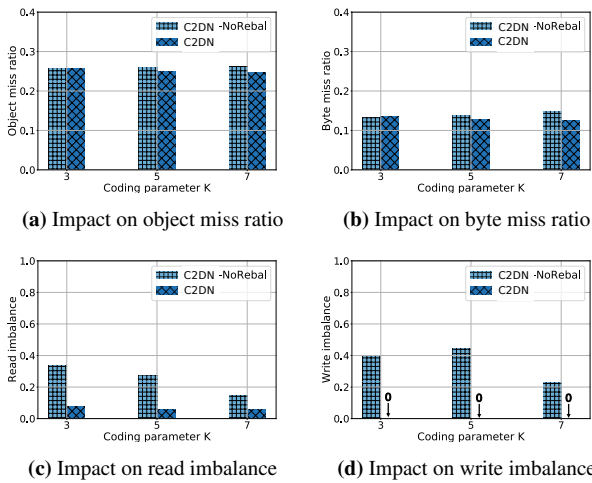
Figure 14: Impact of parameter K on miss ratio and load imbalance.

(a) Impact on byte miss ratio

(b) Impact on object miss ratio

(c) Impact on read imbalance

(d) Impact on write imbalance
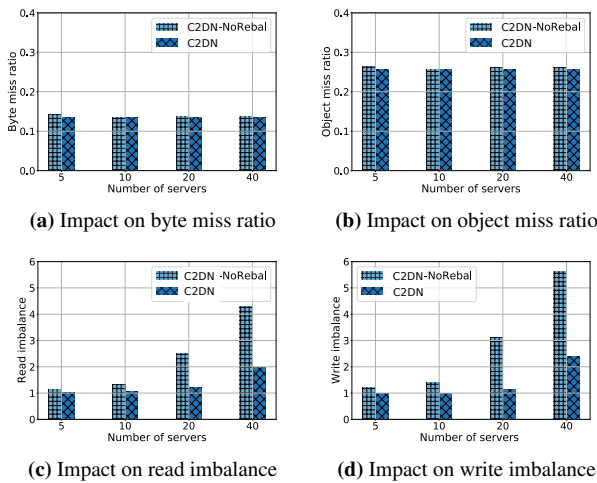
Figure 15: Impact of number of servers on miss ratio and load imbalance.

(a) Object miss ratio

(b) Byte miss ratio

(c) Read load balancing

(d) Write load balancing
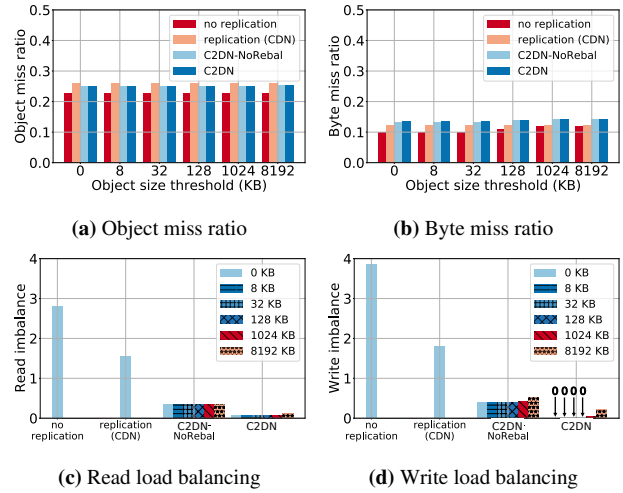
Figure 16: Impact of coding size threshold on miss ratio and load balancing (LRU).

(a) Impact on object miss ratio

(b) Impact on byte miss ratio

(c) Impact on read imbalance

(d) Impact on write imbalance
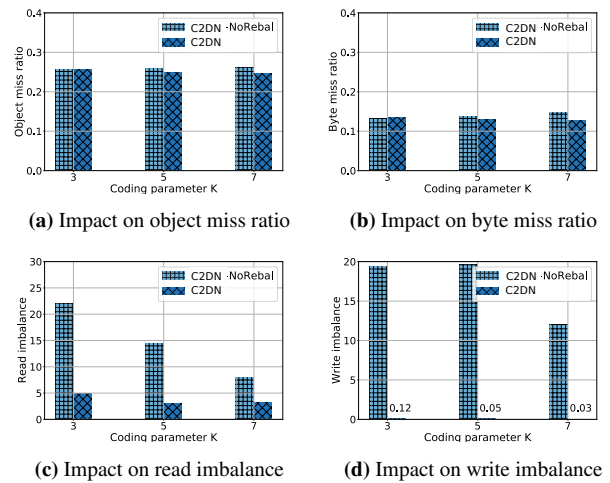
Figure 17: Impact of parameter K on miss ratio and load imbalance (LRU).

Optimizing Network Provisioning through Cooperation

Harsha Sharma,* Parth Thakkar,* Sagar Bharadwaj,* Ranjita Bhagwan, Venkata N. Padmanabhan,
Yogesh Bansal, Vijay Kumar, Kathleen Voelbel
Microsoft

Abstract

The rise of cloud-scale services has fueled a huge growth in inter-data center (DC) Wide-Area Network (WAN) traffic. As a result, cloud providers provision large amounts of WAN bandwidth at very high costs. However, the inter-DC traffic is often dominated by *first-party applications*, i.e., applications that are owned and operated by the same entity as the cloud provider. This creates a unique opportunity for the applications and the network to cooperate to optimize network provisioning (which we term as optimizing the *"provisioning plane"*), since the demands placed by dominant first-party applications often *define* the network. Such optimization is distinct from and goes beyond past work focused on the control and data planes (e.g., traffic engineering), in that it helps optimize the provisioning of network capacity and consequently helps reduce cost.

In this paper, we show how cooperation between application and network can optimize network capacity based on knowledge of the application's deadline coupled with network link failure statistics. Using data from a tier-1 cloud provider and a large enterprise collaboration service, we show that our techniques can potentially help provision significantly lower network capacity, with savings ranging from 30% to 45%.

1 Introduction

The growth of cloud-scale mega services [1] has fueled a huge increase in network traffic, not only within data centers (DCs) and on the Internet but, importantly, also on the inter-DC wide-area network (WAN). We observe that the inter-DC WAN traffic for large public cloud providers is dominated by first-party applications and services e.g., the e-commerce service in the case of Amazon, Google search and Gmail in the case of Google, and Bing search and Office 365 in the case of Microsoft [8, 9]. This presents both a challenge and an opportunity.

The challenge is that, as application usage grows, so does the inter-DC WAN traffic. This leads to increased provisioning of inter-DC WAN capacity, often with a multiplicative factor to provide redundancy. Note that large cloud providers typically build and operate their own inter-DC WAN with owned and leased fiber, which is different from a small provider who might use an ISP for such connectivity. The increased availability of bandwidth, and the fact that it is sunk cost (i.e., already paid for and so might as well be utilized), fuels the appetite of existing and new applications. They consume more bandwidth (though there might be back-pressure due to mechanisms such as traffic engineering [7, 8] or internal pricing), causing the network to forecast higher usage for the future, which in turn leads to increased network provisioning and so forth. Such a vicious cycle is quite expensive since inter-DC WAN bandwidth imposes an amortized annual cost of 100s of millions of dollars on a large cloud provider [7].

This situation is quite different from the third-party application setting, wherein market forces of cost and price operating between the consumer of bandwidth (third-party application) and the provider of bandwidth (cloud provider) would tend to keep the process in check. If the provider has excess capacity, they could find new customers to sell it to rather than encouraging existing customers to be profligate in their use of bandwidth. There is an opportunity to do better in the first-party case by leveraging *cooperation* across the network and the applications. Specifically, rich information flow between the application and the network would enable informed network provisioning. We term the task of provisioning network capacity as optimizing the "provisioning plane", to set it apart from the well-established notions of the control and data planes.

We leverage the knowledge of the *application's deadline* to optimize network provisioning. Such an application deadline is quite distinct from the more common notion of delay tolerance, which centers on the latency that the application can tolerate in network communication. Application deadline, on the other hand, arises from the application's ability to pause, or defer, some or all of its activities and the associated com-

---

munication for an extended period of time, for example, when there is a loss of network capacity because of link failures. If the network capacity is likely to be restored (e.g., by repairing the failed link(s)) within the period of the application deadline, then we can dispense with the provisioning of redundant network capacity for "deferrable" traffic.

If user impact is to be avoided, the deferrable application activity, and the resulting traffic, should be in the background. A good example is load balancing, which might be triggered when a server is at the risk of running "hot" on one or more resources such as CPU, IO (I/O operations) capacity, or storage space, and therefore needs to shed load. Load balancing would typically involve the transfer of a large volume of data (e.g., user mailboxes in the context of an email application). This activity could be paused for a length of time so long as there is enough "headroom" in resources, i.e., none is on the verge of being saturated. Based on past data, we can make a conservative but specific choice of deadline to facilitate network provisioning.

We develop a generic framework for applications to express their demands in a way that reflects the traffic volume, deadlines, and desired probability of satisfaction (i.e., the likelihood that the demand will be met). We also develop a model for link failures and repairs that is informed by historical data. A key aspect of this model is the data-driven discovery of links with correlated failures, say because they share one or more components (e.g., fiber conduit, power supply, etc.) and the recreation of such correlated failures to simulate scenarios that are particularly challenging from the viewpoint of capacity provisioning.

We evaluate the above using data from the WAN of a tier-1 cloud provider, Microsoft, and from a large enterprise collaboration service, M365 Substrate (which we shall hereafter refer to as Substrate), which uses this WAN. We find that application-informed provisioning reduces network capacity by more than 30% in multiple regions.

Note: While we use data from commercial services — Microsoft and Substrate— to drive our analyses here, this data is highly sensitive due to commercial reasons. Therefore, we are not in a position to report metrics such as the traffic volume, network capacity, failure characteristics, etc. in an absolute sense and report relative numbers instead.

1.1 Comparison with Traffic Engineering

Before proceeding, it is useful to compare our work on cooperative provisioning with the well-established prior work on traffic engineering.

Traffic engineering focuses on supporting a specified demand (i.e., source-destination flows, quality of service requirements) on a given network (i.e., the network topology, including link capacities). Both the demand matrix and the network topology are taken as input and traffic engineering looks for ways of supporting the demand at *run time* through

techniques such as routing and path selection [8], smoothing to shift traffic peaks into the valleys [7], and using store-and-forward techniques to deal with temporal offsets between traffic peaks in different parts of the network [13]. There is a body of traffic engineering work that specifically considers the problem of satisfying deadlines in the face of new demands, link failures, etc., by employing admission control, online scheduling, and fairness across demands when the deadlines cannot be satisfied [11, 22].

In comparison, our work on cooperative provisioning focuses on the *planning phase* that precedes the creation of the network or the augmentation of its capacity. This requires modeling link failures upfront by simulating the failures of (combinations of) links and making sure that sufficient capacity is provisioned to accommodate the demands even in the face of such failures. In contrast, traffic engineering deals with link failures as these arise and does not entail provisioning capacity. Furthermore, in optimizing the provisioning of network capacity, the ability to defer traffic by pausing workload, for days or weeks, provides us a qualitatively greater flexibility for optimization than the QoS requirements that are typically dealt with in traffic engineering [11, 22]. For instance, the former would allow tiding over link failures without redundant provisioning, while the latter typically would not.

The opportunity to optimize provisioning by counting on the ability to pause certain demands arises because of the first-party setting, which enables cooperation between the network and the applications. Therefore, cooperative provisioning is limited to settings where such cooperation is feasible. Traffic engineering, on the other hand, is applied more broadly (e.g., in ISP networks) and as such cannot assume such cooperation.

2 Application Traffic Demands

The starting point for network capacity provisioning is the demand placed by applications. Since building a network or augmenting the capacity of an existing network would typically take time (several months to more than a year), there is also the need to *forecast* future demand. Forecasting is a well-studied problem and there exist many techniques for it [15, 19]. Thus, in this paper, we do not focus on forecasting and our analysis is based on taking a snapshot of the present demand and using it to perform capacity planning, with and without our optimizations. However we do show in Section 4.3 that our findings carry over even if applied to future demand obtained through forecasting.

2.1 Demand Specification

An application, i, specifies its demands as a set of discrete time series, with the jth demand on behalf of i being specified as $D_{ij} = (t, A, B, V, d, p)$, i.e., a demand for conveying a volume V of traffic from location A to location B, expressed at time

t and with a deadline of d (that is, the demand should be satisfied during $[t, t+d]$), and with the desire that the demand be satisfied with a probability of p, i.e., in the network failure scenarios that cumulatively account for a fraction p of time. (Each of these quantities is set by the application i and should carry the subscript ij, but we drop it for the ease of exposition.) We describe two types of application demands which are relevant to this paper: immediate and deferrable.

Immediate: Such traffic tends to be in the critical path of user latency, so there is little temporal flexibility. The deadline for such a demand is "now", so the demand can be expressed as a data rate to be supported between a specified source and destination. Hence we can simplify the formulation of the demand as $D_i = (t, A, B, r, p)$, where we subsume V and d by the rate $r = V/d$. Note that if the network were provisioned to support the desired rate r for this demand continually, we would be able to transfer volume V within a deadline d.

Deferrable: The traffic demand arising from asynchronous activity tends to be temporally flexible, i.e., deferrable, with a long deadline. A good example is traffic arising from distributed load balancing, wherein resources on a server (e.g., CPU, IO capacity, storage space) grow in terms of utilization (i.e., start becoming "hot"), necessitating the rebalancing of the workload and entailing the transfer of related data (e.g., a user's mailbox in the context of an email service or folder in the case of a storage service). The urgency of the load balancing activity and hence the deadline of the resulting traffic demand would depend on how much headroom there is on the server resources that are heating up.

Deferrable demand would be specified with a deadline d that reflects the degree of temporal flexibility. For instance, in the case of load balancing triggered workload, there might be enough headroom in the server resources (i.e., none is close to being saturated) to allow d to be set to days or even weeks. Note that the demand can tolerate such latency (e.g., by slowing, pausing or turning off the application components responsible for the demand); however, once the application component is resumed and it actually starts its data transfers, these would be completed in a much shorter and typical "network timescale" (e.g., within seconds or minutes or hours, depending on the size of the transfer). Also, just because a demand can be temporarily paused, it does not mean that it can be forgotten. So, the network would be provisioned with sufficient capacity to allow "catch-up" on the deferred demand once it is resumed. Processes that perform periodic tasks on data such as garbage collection, compliance checks, and workload analytics can also cause asynchronous traffic demands.

3 Cooperative Provisioning

In this section, we describe how the network, with the application's cooperation, can provision the network efficiently. We

first provide some background on network provisioning and how it is currently done. Next, we discuss two specific opportunities for cooperative provisioning: (a) deriving a smoothed demand signal based on explicit application input, and (b) provisioning network redundancy in a way that is cognizant of the demand deadlines and the repair time of links. Finally, we describe a mathematical framework which, using constraint optimization, ensures appropriate provisioning of network capacity, including redundant capacity, to satisfy all demands. We dub this framework Approv, short for "Application-informed Provisioning".

3.1 Background on Capacity Provisioning

We sketch how capacity provisioning is done by the cloud provider, Microsoft. Although we do not have information from other providers, we believe that the process outlined here is general and not provider-specific.

Periodically, e.g., every few months, the cloud operator forecasts demands between each datacenter (DC) pair and subsequently provisions network capacity to satisfy all demand forecasts. Since the demands arise from third-party applications (which the operator has no real leverage over) as well as first-party ones, the network typically works with just the actual traffic originated by applications (i.e., an implicit signal) rather than an explicit expression of application traffic demand (e.g., as in Section 2.1). Such an implicit signal is derived from application traffic categorized into tiers of service, with each tier corresponding to a different priority level [7, 8].

Based on this implicit demand signal derived from actual traffic, the operator derives the peak or 95%ile (P95) traffic level and uses this to forecast the traffic level, say months or even more than a year into the future. In the absence of any additional context from the applications sourcing the traffic, the operator has no choice but to work with the implicit demand signal, no matter how spiky it is (i.e., how high the peaks are).

With the demand so determined, the operator proceeds to simulate the failure of various links and combinations of links. A simulator is used to route each demand over possibly multiple paths chosen based on the latency and the available bandwidth. If the demands cannot be satisfied using the available bandwidth, the operator would augment capacity in the network (i.e., add redundant capacity) so that the demands are satisfied even in the face of such failures. Specifically, the capacity of a link is augmented if its utilization during simulation rises above a high-water mark and it is reduced if the utilization drops below a low-water mark. At the end of this iterative process of simulation, the operator would arrive at the *network capacity build out plan*, specifying the number and capacities of the links needed.

There are a couple of details worth noting. First, in general, the operator would start with an existing network and then determine the capacity augmentation and link decommission-

ing needed based on the demand forecast. However, to enable fair comparisons in this paper, we provision the network from scratch, i.e., starting with links of zero capacity, which helps avoid the need to carry the baggage of past provisioning done without the benefit of Approv. Second, for simplicity, we work with the specified topology (where each link starts out with zero capacity) and only compute the capacity needed on the links that are part of the topology. While the Approv framework is general enough to accommodate the creation of new links, the ability to create a link between two locations would, in general, be constrained by practical considerations that would have to be provided as additional input.

3.2 Leveraging Application Cooperation

We can leverage the application's cooperation — specifically that of first-party applications that have nothing to hide from the network — to improve the network capacity provisioning process outlined above in two ways.

3.2.1 Explicit Demand Signal to Aid Smoothing

First, rather than just work with the implicit signal of actual traffic sent, the operator can take advantage of application demands expressed explicitly per the framework presented in Section 2.1. Such explicit knowledge would enable the operator to calculate the smoothed demand.

For example, even if the peak (or P95) rate of traffic sent by an application was 1Gbps, knowledge of the explicit application demand (including the deadline d from Section 2.1) might allow the operator to determine that the application's traffic could have been smoothed down to a peak rate of 0.8 Gbps. Therefore, instead of basing its forecast, and consequently the actual capacity provisioning computation, on the 1Gbps peak, the operator could work with the smoothed 0.8 Gbps peak, thereby "rightsizing" the capacity. In the case of a deferrable demand, expressing the demand in terms of the volume V and deadline d would enable further rightsizing of capacity compared to just smoothing down the demand expressed as a rate r. The reason is that much or all of a demand in terms of V and d could be fit within the valleys and headroom of the immediate demands (Figure 2), obviating the need for any additional provisioning.

3.2.2 Rightsizing Redundancy for Deferrable Demands

Second, knowledge of the deadlines enables the operator to "rightsize" the redundancy too. The intuition is that if an application demand has a longer deadline than the time to recover a failed link, then during capacity provisioning (and during the subsequent operation of the provisioned network), we can assume that part or all of such demands are simply paused or deferred until the failed link has been repaired. This would avoid the need for provisioning full redundancy for such deferrable demands.
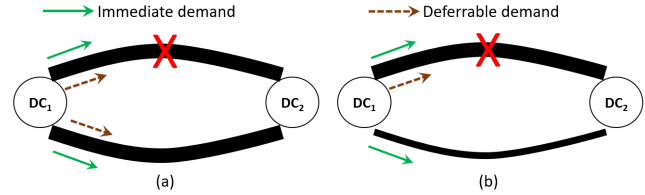
Figure 1: Rightsizing of redundancy provisioning: (a) shows the default provisioning of redundancy with no distinction between immediate and deferrable demands. Upon failure of the primary link at the top, the backup link at the bottom has the capacity to carry the full load of traffic). (b) shows how the provisioning of redundancy reduces by confining it to the immediate demand. The thinner backup link is sized to carry just immediate demand, although the overall provisioning would be sized to accommodate the "catch-up" of the deferred demand upon repair.

A simple example illustrates the savings made possible by such informed redundancy provisioning. Consider a network comprising just two nodes, A and B. Consider two cases:

Case I: The application's implicit demand is 1 GB/day.

Case II: The application explicitly specifies a demand of $V = 30$ GB with a deadline of $d = 30$ days, which also works out to 1 GB/day.

Say the cloud operator knows that the $A - B$ link could be down for up to 10% of the time, i.e., for up to 3 days in the month. Therefore, to support the implicit demand in Case I, the operator would have to provision at least one additional link of 1 GB/day capacity (and possibly more depending on the likelihood of concurrent failures of multiple links). Therefore, the operator would have to provision a total of at least $1 + 1 = 2$ GB/day capacity, and possibly more, on the $A - B$ path.

On the other hand, in Case II, explicit knowledge of the deadline would enable the operator to determine that the demand could be paused, or deferred, temporarily to "tide over" the network link's downtime. Of course, once the link has been restored, there would be the need to "catch up" on the deferred demand. Still, given the up to 10% downtime of the link, the operator can get away with provisioning a single $A - B$ link of capacity 1.11 GB/day (computed as 1/(1-0.1)), which would be significantly lower than the (at least) 2 GB/day in Case I above. (In this illustrative example, we ignore the quantization of bandwidth, i.e., the minimum step size for allocation.)

3.2.3 Summary

In general, we would have a mix of "immediate" user-facing demand and "deferrable" background demand, and the former cannot tolerate any delay. Therefore, we might still have to include redundant links. However, by rightsizing the provisioning of redundancy for the deferrable demand, we would

be able to reduce the overall capacity provisioned, as illustrated in the simple example in Figure 1. Furthermore, as explained above, specifying deferrable demand in terms of volume and deadline would enable more effective provisioning (perhaps even fitting within the valleys and headroom of the immediate demand) than working with a smoothed rate.

In the remainder of this section, we formalize our framework for cooperative capacity provisioning and also present our method for simulating network link failures and repairs (including the concurrent failure of links that carry shared risks), informed by the history of actual link failures and repairs.

3.3   Framework for Capacity Provisioning

Our Approv framework models network provisioning as a constraint optimization problem using a linear program (LP). As with any network provisioning approach (Section 3.1 provided some background on this), the LP needs to simulate all "likely" link failures, singly or in combination, and ensure that the network has sufficient redundant capacity to fulfill all demands despite such link failures.

A key challenge in our work arises from our richer demand model compared to prior work. State-of-the-art approaches [1] take demand data rates as input (e.g., 1 Gbps from A to B) and then only simulate link failures as point-in-time events, to verify the satisfaction of demand in the face of failures. Since we support demands that specify deadlines (d) that could stretch to days or even weeks, we incorporate a much richer notion of failures, including the actual duration, i.e., the time from the loss of capacity due to the onset of a failure episode until the restoration of capacity upon repair. As we discuss, this approach enables optimizations that are not possible with point-in-time simulation of failures.

Given a time-window (t_s, t_e), We define a *failure scenario* as a set of per-link time-series $\{f_1, f_2, \ldots, f_n\}$, where n is the number of links in the network. f_l is a time series representing the status of link l, i.e., whether it was up (working) or down (failed), over time. f_l is a time series, $f_{l1}, f_{l2}, \ldots, f_{lt}$ over the full duration of the simulation, where $f_{lt} \in \{0(down), 1(up)\}$ and time is discretized (in steps of 1 hour in our work). To synthesize realistic failure scenarios that extend over the full duration of the simulation, we have built a novel history-based failure model, described in Section 3.4.

The Approv capacity provisioning formulation assumes a fixed network topology, where the nodes are either datacenters in the cloud network or network points of presence. Given such a topology and the application demands the Approv provisioning framework allocates capacity to each link in the topology so as to satisfy all demands, by simulating two functions:

• *Topology and Routing:* Approv incorporates constraints arising from the network topology and the set of valid network-level routes between any two datacenters (DCs). The routes
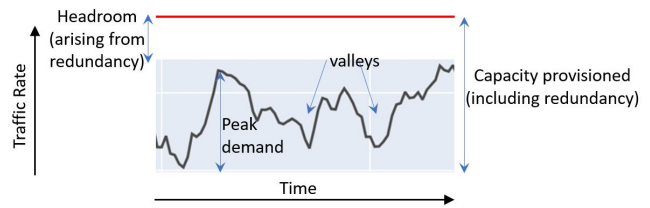
Figure 2: Illustration of the peaks and valleys in the time series of the immediate demand, and of the headroom above the peak, arising from the overall capacity provisioned, inclusive of the redundancy. The valleys and the headroom represent capacity that could be used to accommodate background demand, subject to its deadline.

are derived from the large inter-DC WAN deployed by Microsoft.

• *Link Failures:* Approv also incorporates constraints to capture the effect of each failure scenario that is simulated. This helps ensure that the network is provisioned with enough capacity to fulfill all demands even in the face of such failures.

For ease of exposition, we use a uniform p (i.e., probability of satisfaction) for all demands. Later, in Section 3.3.2, we relax this assumption to generalize the framework to incorporate a demand-specific p.

Given a set of failure scenarios $\mathbf{F}$, we construct a "capacity provisioning LP" with constraints corresponding to each combination of demand and failure scenario. The LP generates a capacity provisioning plan in two steps:

1. *Provisioning of immediate demands:* Such demands have an immediate deadline, so we represent these in terms of the rate r, as discussed in Section 2.1. The demanded rate needs to be supported even in the face of the failure scenario considered. To ensure this, we sweep across time, from the start to the end of the simulation. We ensure that the network is sufficiently provisioned to support the immediate demands at each point in time, which corresponds to a specific combination of link failures.

2. *Provisioning of deferrable demands:* Deferrable demands are expressed in terms of the desired volume, V, of data to be transferred and the corresponding deadline d. Our provisioning framework first checks to see whether (and if so, how much of) such demands could be accommodated in the valleys and the headroom of the capacity provisioned above for the immediate demands (see Figure 2). To the extent the deferrable demand exceeds what can be so accommodated, the framework augments the capacity on one or more links to ensure that the deferrable demands are satisfied too (in addition to the immediate demands). In doing so, we take advantage of the demand smoothing and redundancy rightsizing techniques discussed in Section 3.2.

At the end, we arrive at the network build plan, which gives the capacity to be provisioned on each link in the network. We present the LP formulation and an example in Appendix A.1 and A.2.

3.3.1 Accommodating Probability of Satisfaction (p)

To ensure that the capacity provisioned in the network build plan satisfies the demand with the desired probability p, we use the failure model (discussed next in Section 3.4) to generate a large number of failure scenarios. As discussed in Section 3.4, the generation of failure scenarios is governed by the failure and repair history of each link and group of links. Consequently, the more likely link failure combinations will appear more often in the failure scenarios, just as we would desire. Since we would like the network to be provisioned so as to satisfy the demand with probability p, or equivalently in at least a fraction p of the failure scenarios considered, we sort the $|\mathbf{F}|$ individual failure scenarios in increasing order of the capacity of the build plan generated when each such scenario alone is simulated. To enable the sorting, we consider the impact of each failure scenario on the build plan capacity in isolation instead of the collective impact of a set of failure scenarios, as in the capacity provisioning LP discussed in Section 3.3. We then consider just the first p fraction of the failure scenarios (i.e., the $p.|\mathbf{F}|$ scenarios with the least impact on capacity) and disregard (or cut off) the last $1 - p$ fraction of scenarios (i.e., the ones with the greatest impact on capacity). This subset of failure scenarios is then provided as input to the capacity provisioning LP to generate the build plan.

3.3.2 Accommodating Demand-Specific p

In general, each demand D_i could have its own p_i, representing the desired probability of satisfaction for that demand. To accommodate such demand-specific levels of p, we employ an iterative process. We first sort the p_i in increasing order, i.e., going from the least level of assurance sought by a demand to the highest. For ease of exposition, we assume that the sorted order is $p_1, p_2, ..., p_n$. Then, we proceed as follows, where in each subsequent step, the additional capacity provisioned, if any, is over and above that which was already provisioned in the preceding steps. In other words, the capacity provisioned never decreases as we progress through the steps:

1. First, present all demands $(D_1, D_2, ..., D_n)$ to Approv and sort the failure scenarios in increasing order of their *individual* impact on capacity (as outlined at the end of Section 3.3 above). Then, pick p_1 as the cutoff in this sorted list of failure scenarios (i.e., focus on just the first p_1 fraction of the failure scenarios) and run Approv on these, to arrive at the *cumulative* capacity. This would ensure that the network is provisioned to satisfy all demands with probability (at least) p_1.

2. Then, exclude the first demand, D_1, and present the rest $(D_2, ..., D_n)$ to Approv and sort the *remaining* failure scenarios (i.e., excluding the ones already satisfied in step 1 above) in increasing order of capacity impact. Then, pick p_2 as the cutoff in this sorted list of failure scenarios, and proceed as in step 1 above. This would ensure that demands $(D_2, ..., D_n)$ are satisfied with probability (at least) p_2. Note that the provisioning performed in step 1 has already ensured that all demands $(D_1, D_2, ..., D_n)$ are satisfied *concurrently* in the face of the p1 fraction of failure scenarios. It is only for the additional p2 - p1 fraction of failure scenarios (during which D1 need not be satisfied) that some of the capacity provisioned to satisfy D1 in step 1 could potentially be used to satisfy $(D_2, ..., D_n)$. Therefore, this subsequent step (step 2) does not *impinge* on the provisioning done in the earlier step (step 1).

3. Proceed accordingly, progressively raising the bar on the probability of satisfaction while narrowing down the corresponding set of demands considered at each step.

4. The process would conclude when, in the last step, we only consider demand D_n and ensure its satisfaction with probability p_n.

This ensures that the provisioning at the end of the process would satisfy all demands D_i with the corresponding probability p_i.

3.4 Failure Modeling

In this section, we describe our *history-based* failure model that we use to generate realistic failure scenarios to drive the provisioning framework described in Section 3.3. Using a history of link failure characteristics, we build a generative model of link failures that captures characteristics such as the distribution of Time To Recovery (TTR) and the Time Between Failures (TBF) for the individual links, and the correlation of failure and recovery across links.

Dataset: The dataset we use to build the failure model contains detailed link failure data at hourly granularity collected over a period of 13 months for the over 500 links in Microsoft inter-DC WAN. The WAN consists of 17 interconnected regions, such as Asia-Pacific (APAC) and North America (NAM), with each region including multiple datacenters. Each link l in the network is characterized by a discrete *link-status vector* time-series f_l, where f_{lt} is 1 if the link l was up at time t and 0 if the link was in a failed state at time t.

We build a separate model for each region to keep the modeling tractable, since there could be correlations — accidental or otherwise — across arbitrary pairs of links. Accordingly, our failure modeling uses two main steps, *Link Clustering and Characterization* and *Failure Scenario Generation*. We now describe each step.

Cluster no.	Correlation	% links	Cluser no.	Correlation	% links
0	0.81	46.3	1	0.75	7.3
2	0.79	7.3	3	0.79	7.3
4	0.80	4.9	5	0.92	7.3
6	1.0	2.4	7	1.0	2.4
8	1.0	2.4	9	1.0	2.4
10	1.0	4.9	11	0.97	4.9

Table 1: Avg intracluster Pearson correlation coefficient

3.4.1 Link Clustering and Characterization

Link failures are often correlated since links could share components such as a power source, a router, or a fiber cable [16, 20]. To capture such correlations, we cluster links that display similar failure patterns in our data. We run the Complete-linkage agglomerative clustering algorithm [3] using the *1-Pearson correlation coefficient* between the link-status vectors noted above as the distance metric between two links to form such clusters of links. Links which fail at the same time will have correlated failure patterns and hence land in the same cluster (In Section 3.4.2, we explain how we simulate failure of all links in a cluster simultaneously). We evaluated the average of Pearson correlation coefficient between links within each cluster, while sweeping over the count of clusters, and determined that setting the number of clusters to 12 yielded a satisfactory division of links, with the average intra-cluster Pearson coefficient being 0.9.

Table 1 shows average intra-cluster Pearson correlation coefficient for the Asia-Pacific (APAC) region. The high intra-cluster Pearson correlation coefficient underscores the high degree of correlation in the failure pattern of links within the clusters. Note that cluster 0, containing over 46% of the links, comprises links whose failure pattern does not correlate with that of any other link; in fact, these links suffer few failures and so their link status vectors are set to 1 at almost all times. So, this is not considered as a failure cluster during the generation phase (Section 3.4.2).

Since the clusters capture links with similar link-status vectors, we assume that links belonging to a cluster have the same distribution of Time To Repair (TTR) and Time Between Failures (TBF). Accordingly, we estimate a single distribution of TTR and TBF for the links in each cluster. From the link-status vectors in a given cluster, we gather all the link TTRs and estimate the Cumulative Distribution Function (CDF) of TTR by using linear interpolation. We do likewise to estimate the CDF of the TBF. We then use these CDFs to generate plausible failure scenario, i.e., a set of realistic link-status vectors for each link.

3.4.2 Failure Scenario Generation

Our failure scenario generation algorithm uses the estimated CDFs to generate a common link-status vector for each link cluster for a given time-window. This approach assumes that links within a cluster are perfectly correlated: all links fail and are restored at exactly the same time. While simplifying
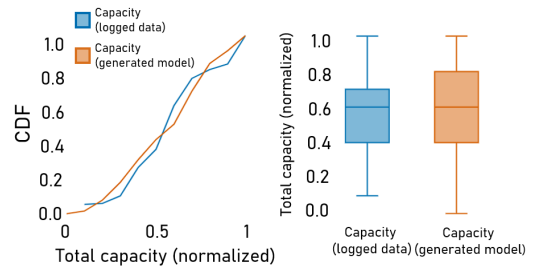
Figure 3: Comparing distribution of total capacity between logged historic data and generated failure scenarios.

the task of generating failure scenarios, this assumption of perfect correlation also ensures that the network provisioned in Section 3.3 is "stress-tested", i.e., made tolerant to a worst-case scenario where a chunk of capacity on correlated links is lost in one fell swoop.

Algorithm 1 Algorithm to generate timeseries of failures.

$uf \leftarrow$ uptime fraction
$ttr\_cdf \leftarrow$ CDF of TTRs from past data
$tbf\_cdf \leftarrow$ CDF of TBFs from past data
$timesteps \leftarrow$ no. of timesteps for which we generate timeseries

$timeseries = \emptyset$
Add 1 or 0 to $timeseries$ with probability uf or $1 - uf$ respectively
$t = 1$
while $t \leq timesteps$ **do**
    if last item in $timeseries = 1$ **then**
        $tbf =$ Sample a value from tbf_cdf
        Add tbf 1s to $timeseries$ followed by a 0
    else
        $ttr =$ Sample a value from ttr_cdf
        Add ttr 0s to $timeseries$ followed by a 1
    $t =$ Length of $timeseries$
Output: First $timesteps$ elements of $timeseries$

We use Algorithm 1 to generate a common link-status vector per-cluster. A value of 1 at $timeseries[t]$ denotes that all the links in the cluster are up at timestep t. Similarly, a value of 0 denotes that all the links in the cluster are in the failed state at timestep t. If the latest state of the generated timeseries is 1, we determine how much longer the links in the cluster will stay up by sampling a value, tbf, from the TBF distribution. Similarly, if the latest state is 0, we sample ttr from the estimated TTR distribution and fail all links in the cluster for the next ttr timesteps.

3.4.3 Properties of generated timeseries

We inspected some properties of the generated link failure timeseries and compare it with those of the actual failures recorded in the history.

Total network capacity available in a region at a given time is the sum of capacities of links that are up at that time. Therefore the total network capacity varies with time. Figure 3 shows the CDF and Box plots of the distribution of total network capacity in a region and compares the distributions that we see in the past data and the generated failure scenarios. 10 failure scenarios each spanning a month were generated from the past 13 months of logged data. The capacities are min-max normalized using the same minimum and maximum normalization factors for both generated and logged capacities. The generated timeseries has a distribution of total capacity that is reasonably close to the distribution we see in the past data, but does not exactly replicate it. In particular, the minimum generated capacity is lower than the minimum logged capacity. This is because, as noted above, when failure scenarios are generated, all links in a cluster are failed together, thereby generating worst case failure scenarios that may not have occurred in the past. This ensures that enough capacity is provisioned for scenarios not present in our logs but are nevertheless possibilities because of failure correlations and so would be prudent to be prepared for.

3.5 Provisioning and Control Interfaces

In this section, we describe the APIs required in the provisioning and control plane between the applications and the network to realize the gains from cooperative provisioning.

Every few months, each application component provides its immediate and deferrable demands between every DC pair as a time-series to the network. Figure 4 provides an example of how applications specify their demands to the network provisioning process. Immediate demands specify an hourly rate r while deferrable demands specify a volume V of traffic that is generated within each day. Additionally, for each application, the deferrable demand includes a deadline d and a probability of satisfaction p.

Immediate Demands →

Time	From	To	Application	Rate (Gbps)
1-Jun-21 12:00:00	SIN	HKG	Mail Client	3000
1-Jun-21 12:00:00	SIN	HKG	Search	1000
1-Jun-21 13:00:00	SIN	HKG	Mail Client	2500

Deferrable Demands →

Time	From	To	Application	Volume (Tb)
1-Jun-21 12:00:00	SIN	HKG	Mailbox Load Balance	1500
1-Jun-21 12:00:00	SIN	HKG	Server Initialization	300
1-Jun-21 12:00:00	SIN	HKG	Periodic Tasks	50
2-Jun-21 12:00:00	SIN	HKG	Mailbox Load Balance	1200

Application	d	p
Mailbox Load Balance	7days	0.999
Server Initialization	7days	0.99

Figure 4: The specification of immediate demands using rate and deferrable demands using volume.

Choosing deadlines: In some cases, component owners can systematically calculate deadlines, while in others, it is left to domain experts to use their judgement and specify the deadline. Section 4.1 provides examples from the Substrate service that fall into both categories. To gain more insight into the choice of deadlines, we interviewed the service owners, who are the domain experts. Since the service owners are going from operating in a mode without any deferrable traffic to one where they are willing to defer part of their traffic to enable cost savings, it is understandable that they were conservative in picking appropriate deadline values. Also, in some cases, they prefer the component to have a minimum amount of capacity available at all times. To support this, we simply divide the component's WAN traffic demands into an immediate component and a deferrable component. For instance, for the mailbox load balancing component described in Section 4.1, the domain experts require 10% of traffic to be flagged as immediate.

Forecasting: The network's provisioning process takes these inputs and forecasts future usage. To do this for immediate demands, it first aggregates hourly usage across all applications, and then computes the P95 usage over an extended period such as a day. Using this number across multiple days, it then determines a trend and forecasts future usage. For deferrable demands, the network forecasts daily volume of traffic for an extended period, say months, and inputs these to Approv. Appendix A.1 explains how Approv uses this input.

Run-time Interface: Since cooperative provisioning is done assuming that part of the demand is deferrable, cooperation of both the network and the application at run-time becomes necessary to live up to this assumption. Specifically, the application and the network need to react quickly and appropriately to link failure events. When a link fails, the network informs deferrable application components to either slow their rate of generating traffic or to pause such traffic altogether. We achieve this through a combination of two techniques: one enforced by the network and the other effected through application cooperation.

When traffic generated by a deferrable component is uniquely identifiable through network-visible identifiers, such as IP addresses or five-tuples, the network uses a bandwidth enforcer similar to previous work [12, 14] to apply backpressure on the application. When it detects a link failure, the bandwidth enforcer reduces the allocation to the deferrable component that needs to be slowed down. Substrate's deferrable components (such as load-balancing, database migration, and background cleanup processes) are designed to slow down when the bandwidth enforcer reduces their allocation. As a result the components either reduce the amount of work they do or pause it altogether. When the failed link comes back up, the bandwidth enforcer increases the allocated bandwidth and sends an explicit signal to the application components to resume normal activity.

Often, however, multiple application components use the same network identifier to send traffic and consequently, network bandwidth enforcement cannot isolate traffic from the individual application components to be slowed down or paused. Fortunately, Substrate uses a job scheduler that controls the progress of background and asynchronous tasks. Hence when

the network informs the application of a link failure, the application's job scheduler explicitly slows the appropriate deferrable component(s), thereby keeping the application's network traffic in line with the reduced network capacity.

We are in the process of productizing WAN capacity provisioning at Microsoft based on Approv. This work is in collaboration with both the WAN team and the owners of the Substrate service. The implementation comprises Approv-based optimization of the offline provisioning pipeline and run-time adaptation . The latter comprises two parts. The first is bandwidth throttling enforced by the network, when there is loss of capacity due to link failure, to limit the traffic of applications that had marked part of their demand as deferrable during the offline provisioning phase. This ensures the protection of the network from overload during such times of capacity crunch, regardless of application actions. The second is run-time adaptation of the Substrate service, to ensure that the application throttles its activity, and hence traffic, in a manner that avoids user impact, e.g., by pausing deferrable components but not the user-facing ones. For this purpose, we have implemented interface enhancements using approximately 2500 lines of C# code, and are running comprehensive tests to verify that all Substrate's deferrable components are indeed able to pause or slow down when required.

4   Evaluation

In this section, we evaluate the benefits of cooperative provisioning. Our evaluation uses Microsoft inter-DC WAN topology and link failure characteristics, and applies Approv to satisfy demands for Substrate, a large-scale service that supports several collaboration applications such as email, shared file services, and enterprise analytics. Substrate accounts for well over a third of the overall traffic on Microsoft WAN and as such plays a significant role in defining the WAN capacity. Therefore, we believe it is useful to analyze capacity even just in the context of the Substrate service. We first describe our methodology, and then turn to our results.

4.1   Application Demands

Substrate consists of various *components* which perform different logical functions. Many components are user-facing; for instance, a component that responds to a user's REST API calls to read email. These components create immediate traffic demands. Additionally, Substrate implements a number of components that use the WAN extensively to ensure high data availability, reliability, and performance. These mostly run asynchronously and therefore their traffic demands are mostly deferrable. By interviewing component owners, we have determined the following four components whose application demands can be deferred.

Mailbox Load Balancing (MLB): This component is specific to the email application built on Substrate. With time,

utilization on some servers (measured in terms of CPU usage, storage or IO capacity) can become disproportionately high. To preempt this from impacting user latency, the load-balancing component periodically schedules mailbox moves from heavily loaded to lightly loaded servers. Such moves can be deferred until a certain deadline. To determine the right deadline, the load-balancing team continuously monitors the free and available resources (i.e., the "headroom") on each server, e.g., the free storage available. Based on the rate at which utilization grows on each server and the available headroom on the various resources, the team calculates a deadline by which the load balancing must complete while still keeping the utilization under control and the user latency unaffected. Currently, this deadline is conservatively calculated as 7 days, which allows for occasional unexpected surges in load .

Periodic Tasks (PT): Substrate requires certain maintenance and analytics tasks to run periodically at a daily, weekly, or monthly cadence. These run in the background and perform two main functions. First, they perform data clean-up. For example, one task permanently deletes data items that are marked as deleted. Another task ensures compliance by performing time-driven deletions as mandated by legislation such as GDPR [2]. Second, they perform analytics tasks that extract useful enterprise-specific information from the data, such as generating weekly reports on how an employee splits their time between meetings and focus time. Based on conversations with owners of these tasks, we determined that these demands are deferrable by up to 1 day.

Server Initialization (SI): Substrate is a rapidly growing cloud service. To support this growth, it periodically adds new servers to its datacenters. This growth happens in bursts and is not gradual: a large number of new servers may become available at a particular datacenter in a particular month, and the next set of new servers may land only much later. The new servers are initialized with data from existing servers that may be in other datacenters, triggering large data transfers over the WAN. This workload is deferrable since such initialization does not have immediate, user-visible consequences. On the flip side however, it has to complete within a reasonable time so that the service can start utilizing the new servers, and thereby support growing demand. Per the operations team, 7 days is a reasonable deadline for the demand arising from the (new) server initialization. Although SI-based demands occur only sporadically, the network provisioning has to explicitly consider this demand since the traffic is bursty with a high peak.

Database Geo-Replication (DG): Substrate uses database-level geo-replication to ensure data availability. Changes to any data item are written to a database, and the database transaction log is replayed at three geo-replicated servers. Hence all components that modify data items generate geo-replication traffic. Replication traffic triggered by user-visible updates such as the creation of a new email is treated as immediate, while the rest is treated as deferrable, with a deadline

that is the same as for the component that generates it: 7 days for MLB and SI, and 1 day for PT.

4.2 Evaluation Methodology

We first describe the inputs used to evaluate Approv. Then, we outline the different provisioning approaches we compare Approv with. Finally, we explain the network topology that we provision for and the link failure characteristics Approv uses to simulate failures.
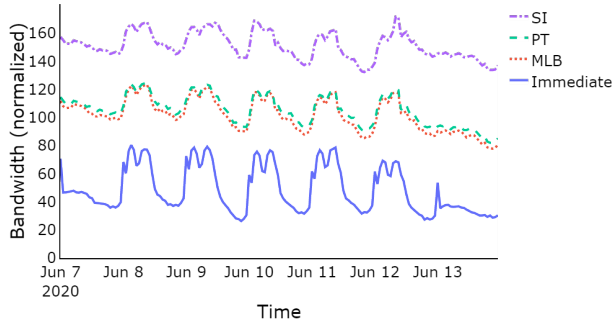
4.2.1 Input Demands

Figure 5: Substrate's network usage over a week in June 2020 on a link from Singapore to Hong Kong, separated into immediate and deferrable (MLB, PT and SI).

We use one month of network usage data from June 2020 to evaluate Approv. The deferrable demand constitutes 59% of total demand by volume, of which mailbox load balancing accounts for about 36%, periodic tasks for about 3% and server initialization for about 20% (see below for elaboration on SI traffic). Figure 5 shows immediate and deferrable demands (stacked over immediate) from a DC in Singapore (SIN) to a DC in Hong Kong (HKG) over one week in June 2020. The curve for each component includes both the traffic arising from updates to the primary replica and the database geo-replication triggered from updates to the secondary replicas.

We did not observe server initialization traffic in June 2020, which is the month for which we have detailed traffic traces. However, to determine the effect it would have on provisioning, we took an estimate of such sporadic traffic, obtained from the concerned team, and overlaid it on top of the total June 2020 traffic. The aggregate traffic traces so synthesized are then provided as input to the provisioning algorithm.

All demands are specified as a time-series in the format shown in Figure 4. Due to the commercial sensitivity of some of this data, as was alluded to in Section 1, we do not spell out the actual values here.

4.2.2 Provisioning Approaches

We quantify the benefits of Approv by comparing it with two alternative approaches:
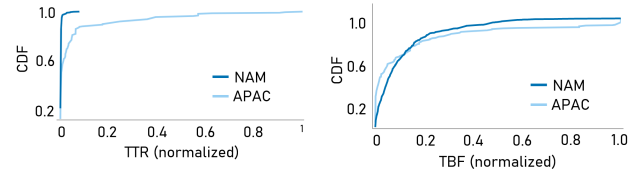
Figure 6: WAN link reliability statistics.

1. Baseline (BL): This is the current state-of-the-art provisioning approach mentioned in Section 3.1. There are several similarities between this baseline approach and the capacity planning scenario described in previous work [1, 21]. All application demands are treated equally. The demands are expressed as rates that need to be satisfied, and are derived by observing the P95 value of rate over the entire dataset. Failures are simulated as point-in-time events during which the demands are to be satisfied, rather than as failure episodes that the demand could tide over.

2. Smoothing-only (SO): This approach performs provisioning with differentiated application demands, but only for smoothing of the deferrable demand (expressed as a rate) to fit in the valleys of the immediate demand (see Figure 2) . In this approach , once the deferrable traffic is smoothed into the valleys, we determine the P95 rate over the dataset and perform provisioning using point-in-time failure simulation. This approach allows us to evaluate how just smoothing traffic , which is well studied in the context of traffic engineering [7, 13], can help improve capacity provisioning.

4.2.3 Network Topology and Link Characteristics

Each provisioning approach mentioned above includes failure simulation. Our failure simulation method is driven by the network topology and link failure data from Microsoft WAN. We concentrate on two regions, Asia-Pacific (APAC) and North America (NAM), since they represent two very diverse network topologies. NAM is the largest region in Microsoft WAN in terms of both the number of links and capacity (it has tens of datacenters and hundreds of links), but consists of mostly land-bound links. APAC, on the other hand, while being smaller, includes an extremely diverse set of links, with a combination of land-bound links and several under-sea cables, given the geography of the region. Figure 6 shows the difference in link failure characteristics between APAC and NAM. While the TTR is almost uniformly low for the links in NAM, there is wide variation in the TTR for APAC, because some links (e.g., those on undersea cables) take time to repair. The TBFs are fairly similar across both regions.

We evaluate each provisioning approach with two failure simulation methods:

1. Replay-based: We simulate failures by replaying the 16-month link-status vector history from Microsoft available to us, with each month treated as a separate failure scenario,

Failure Method	Region	SO (% savings)			Approv (% savings)		
		MLB	+PT	+SI	MLB	+PT	+SI
Replay	APAC	6.5	6.5	9.7	16.8	17.4	38.1
	NAM	9.8	10.4	10.8	28.0	29.2	30.5
Generated	APAC	5.3	5.3	8.2	24.9	25.2	44.2
	NAM	10.8	10.9	11.1	27.3	29.0	31.8

Table 2: Percentage capacity savings over Baseline for SO and Approv, for both replay and failure generation. We first show savings with only Mailbox Load Balancing (MLB) considered deferrable, then we add on Periodic Tasks (PT). Finally, we add Server Initialization (SI) traffic as estimated by the SI team (shaded columns).

yielding a total of 16 scenarios for each of APAC and NAM. *2. Model-generated:* We use the failure model described in Section 3.4 to generate a synthetic but realistic set of link-status vectors.

While the replay-based approach enables evaluation independent of our failure generation model, the latter enables the creation of an unbounded number of failure scenarios (we create 10,000 for each of APAC and NAM), in turn allowing us to evaluate the impact of varying the probability of satisfaction, p (from 0.9 to 0.999).

4.3 Results

In this section, we first evaluate how Approv reduces the provisioned network capacity while supporting Substrate's demand, and improves link utilization compared to the baseline. Next, we perform a sensitivity analysis to evaluate how Approv's savings vary with d and p. Finally, we examine and compare forecasting for the baseline and Approv.

Network Capacity Reduction[2]: We first evaluate the percentage reduction in network capacity provisioned using the three approaches. We calculate the network capacity as the sum of all link capacities in the network. Table 2 shows the percentage network capacity reduction for the Asia-Pacific and North America regions, for both methods of simulating failures: replay-based and model-generated. As noted in Section 4.2.1, since the Substrate components that write to data items trigger database geo-replication (DG), we have included the DG demands in those of the respective Substrate components. Since the historical time-series contains 16 months of data, for the generated method, we generate link-status time-series of length 16 months too.

We concentrate on the last row of the table (NAM with failure generation), though the same explanation applies to the other rows as well. Approv, with MLB alone treated as deferrable, reduces provisioned capacity by 27.3%. With the deferrable set expanded to also include PT, which is a smaller workload and with a tighter deadline of only 1 day, the gains

---

[2]In general, we would also want to consider link-cost-weighted capacity savings, but we defer this to future work.
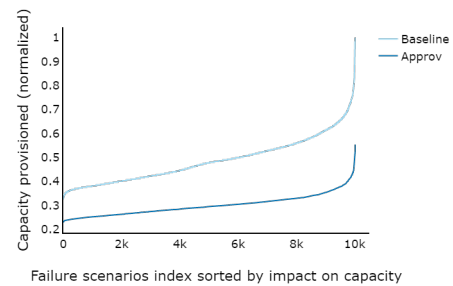
Figure 7: Incremental augments in network capacity for the APAC region for Baseline and Approv.

increase to 29%. Though it is a small increase, there is still value in considering demands like PT as deferrable, since even a small improvement in percentage capacity provisioned yields large absolute gains. Finally, the inclusion of SI as well in the deferrable set adds to the gains, taking it up to 31.8%. The benefits of Approv over SO are also apparent, since SO gives gains of 11.1% whereas Approv's total gains are much higher at 31.8%. Note that both failure simulation methods yield a significant reduction in the capacity to be provisioned, underscoring that the gains are not just specific to our failure model.

Including SI in the deferrable set provided significantly higher gains in APAC than NAM (e.g., an increase of about 20% in capacity savings in APAC compared to just 1-3% in NAM). This is because the number of servers added to APAC and NAM datacenters, and hence the volume of initialization data, was roughly the same in both regions. Since Substrate service is much larger in NAM, the addition contributed less WAN traffic relative to the total in NAM compared to APAC.

Link Utilization Improvement: We now discuss the relative improvement in link utilization due to Approv. We analyze the utilization improvement in APAC network provisioned with Approv using the replay failure scenarios. The relative improvement is computed as 100*(Approv utilization - Baseline utilization)/(Baseline utilization) for each link. Since Approv reduces network capacity significantly, the link utilization improve significantly as well, with an average relative improvement of 42% (note that we compute the utilization improvement on a per-link basis and report the arithmetic average whereas the capacity savings is reported in aggregate).

Sensitivity Analysis: Figure 7 shows 10000 generated failure scenarios for the APAC region on the *x*-axis, sorted by the total network capacity that is provisioned for each failure scenario considered individually. We use all traffic (Immediate, MLB, PT and SI) in this experiment. The graph shows that Approv almost uniformly provisions about 30% less capacity than Baseline. This shows that irrespective of the exact nature of the failure scenarios, Approv consistently helps reduce the network capacity to be provisioned.

Experiment	p=0.999	p=0.99	p=0.9
Baseline	1.00	0.90	0.75
d=12hrs	0.66	0.61	0.51
d=1day	0.62	0.56	0.47
d=2days	0.61	0.51	0.43
d=4days	0.60	0.51	0.42
d=7days	0.60	0.51	0.42

Table 3: The sensitivity of capacity provisioned to the deadline d (varied across rows) and the probability of satisfaction p (expressed in terms of 9s and varied across columns). We normalize all values by the capacity provisioned for the Baseline approach with $p = 0.999$.

Next, we evaluate how Approv's provisioned network capacity varies as we sweep through a range of settings for the deadline, d, and the probability of satisfaction, p. Table 3 shows the results in terms of the normalized capacity provisioned for the baseline case (the first row) and for Approv (the remaining rows).

We see that the capacity to be provisioned decreases as the probability of satisfaction p decreases and also as the deadline d increases. Both of these are as expected since the less "demanding" a demand is, the greater the opportunity Approv has to accommodate the demand without increasing the capacity to be provisioned. However, we also see that the capacity tends to flatten out as the deadline increases, with a diminishing benefit to relaxing the deadline. The reason is that Approv is able to effectively utilize the large amount of headroom in the network arising from the redundant capacity provisioned to support the immediate demand. Note that in the absence of failures, this redundant capacity is not needed for serving the immediate demand and so is available for Approv to accommodate the deferrable demand, even with a shorter deadline.

Table 3 also points to an interesting tradeoff between p and d. For instance, the normalized capacity (0.61) with $p = 0.99$ and $d = $12hrs is roughly the same as it is (0.6) when p is made more demanding ($p = 0.999$) but d is relaxed to $d = $7days.

Forecast: Finally, we evaluate the effect of forecasting error on Approv. We used 3 months of traffic to forecast 10 days of traffic using the Holt-Winter method of forecasting [6] with a seasonality of 7 days. To ensure that the network is able to accommodate sudden, unforeseen peaks in traffic, network forecasts typically use a high confidence interval (CI) envelope, hence, we use a confidence interval of 95% in our experiment. First, we forecast overall traffic, without differentiating immediate and deferrable traffic, as is the current state of the art. Next, we forecast immediate and deferrable traffic separately, as is required by Approv.

For overall demand, the forecast overestimation error for the high end of the 95% CI is 12%. When we separately forecast the deferrable demand, the forecast overestimation error is 11%. In other words, the forecasting is about as accurate for the deferrable part of the traffic as it is for the overall traffic.

Therefore, Approv could as well use the forecast demand as input as it can the current snapshot of the demand (as we do in this paper, using the June 2020 snapshot). Furthermore, given the large amount of headroom that is created by provisioning of immediate traffic (shown in Table 3), we believe Approv has an overall low sensitivity to forecasting errors as well. For these reasons, we believe Approv will remain effective even when used with demand forecasts.

5 Discussion

In this paper, we have used the applications' deadline and desired probability of satisfaction information to provision the network optimally. One challenge that arises from this is incentivizing applications to specify these requirements, which convey their flexibility, appropriately. If the applications are too conservative in specifying their requirements, that would result in reduced capacity savings, whereas if the applications specify their demands loosely, then that might result in an incorrect forecast and hence inadequate network provisioning. We believe that this opens up the possibility of devising pricing mechanisms to encourage applications to specify their demands appropriately, thereby facilitating the cooperative provisioning of network capacity. Another challenge is in extending cooperative provisioning to third-party applications. Unlike with first-party applications, where we assume an implicit trust between the applications and the network that enables free sharing of information up and down the stack, in the third-party setting, we would need to rethink this approach and consider how cooperation can be effected in the absence of such trust.

6 Related Work

In this section, we position our work in relation to previous efforts in traffic engineering, network provisioning and failure characterization.

Traffic Engineering: As discussed in Section 1.1, traffic engineering employs routing and scheduling strategies to satisfy the possibly differentiated demands of applications on a given network. B4 [8] and SWAN [7] use routing algorithms that allow flows to specify different priority levels. Tempus [11] and Amoeba [22] allow applications to set deadlines for traffic, the former uses constraint optimization to satisfy deadlines while the latter uses a graph-based algorithm. Failure-aware traffic engineering has also been explored. FFC [18] splits traffic over multiple paths to be resilient to failures. Song et al. [17] proposed an availability-aware traffic engineering algorithm for optical networks that we believe can be generalized to WANs. NetStitcher [13] uses store-and-forward techniques to deal with temporal offsets between traffic peaks in different parts of the network.

Pretium [10] uses dynamic pricing to route third-party application traffic while handling link failures.

It is important to recognize that traffic engineering and network provisioning are fundamentally different problems. Traffic engineering addresses the short-term, i.e., run-time, problem of how networks route traffic while assuming given link capacities provided as input. Provisioning, on the other hand, addresses the longer-term problem of determining how link capacities should be provisioned in the first place, possibly several months or even over a year into the future, so as to satisfy the communication requirements of applications. In doing so, the provisioning framework (such as Approv in this paper) would invoke traffic engineering techniques under the hood and iteratively, to check if the demand can be satisfied by the network as provisioned and in the face of one or more failure scenarios. If traffic engineering is unable to route the demand, the capacity provisioned would be augmented.

Network Provisioning: We now discuss previous work that directly addresses the problem of network provisioning. Robust network validation [1] proposes a generic optimization framework to determine worst-case network performance across multiple scenarios. They use this framework to model the network provisioning problem which determines the right augments to link capacity so as to handle multiple failures. Liu et al. [21] propose an optimization framework that also solves the network provisioning problem given a set of failure scenarios. Both efforts address network provisioning, however, without using first-party context, and therefore consider all application traffic to be equivalent. Moreover, since they do not inherently consider deferrable demands, they use a simple failure model that only simulates instantaneous, point-in-time failures, with no notion of the temporal dynamics of link failures, i.e., a link failing at a certain time but then being restored at a later time. The baseline provisioning approach we have used in Section 4 is derived from these techniques.

Failure Characterisation: Turner et al. explore and characterize network failures in the CENIC network and provide a methodology to reconstruct these failures [20]. This is similar to the replay-based model we use in Section 4. Shaikh et al. [16] similarly measure link status over time using OSPF link-state advertisements in a large enterprise network. Gill et al. [5] characterize network failures within a datacenter, not on the WAN. Unlike our failure modeling methodology, these efforts analyze historical failures and do not propose generative failure models based on this history. Previous work [4] analyzes optical layer outages in a large backbone and shows that Q-drop events are predictive of upcoming failures. We believe that augmenting our failure model with information from the optical layer would be an interesting future direction.

7 Conclusion

We have described how cooperation, cutting across applications and the network, can significantly lower capacity pro-

visioned in inter-DC WANs in a first-party setting. Using co-operative provisioning, we show how we can provision the WAN more carefully while ensuring that application demands are accommodated with the desired satisfaction probability.

References

[1] Y. Chang, S. Rao, and M. Tawarmalani. Robust validation of network designs under uncertain demands and failures. In *USENIX NSDI*, pages 347–362, 2017.

[2] European Commission. 2018 reform of EU data protection rules. https://ec.europa.eu/commission/sites/beta-political/files/data-protection-factsheet-changes_en.pdf. Accessed Sep 12, 2021.

[3] B. S. Everitt, S. Landau, M. Leese, and D. Stahl. *Cluster analysis 5th Edition*. John Wiley, 2011.

[4] M. Ghobadi and R. Mahajan. Optical layer failures in a large backbone. In *ACM IMC*, page 461–467, 2016.

[5] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. In *ACM SIGCOMM*, 2011.

[6] C. C. Holt. Forecasting seasonals and trends by exponentially weighted moving averages. *International Journal of Forecasting*, 20(1):5–10, 2004.

[7] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving high utilization with software-driven WAN. In *ACM SIGCOMM*, page 15–26, 2013.

[8] C.-Y. Hong, S. Mandal, M. Al-Fares, M. Zhu, R. Alimi, K. N. B., C. Bhagat, S. Jain, J. Kaimal, S. Liang, K. Mendelev, S. Padgett, F. Rabe, S. Ray, M. Tewari, M. Tierney, M. Zahn, J. Zolla, J. Ong, and A. Vahdat. B4 and after: Managing hierarchy, partitioning, and asymmetry for availability and scale in Google's software-defined WAN. In *ACM SIGCOMM*, page 74–87, 2018.

[9] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu,

J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a globally-deployed software defined WAN. In *ACM SIGCOMM*, page 3–14, 2013.

[10] V. Jalaparti, I. Bliznets, S. Kandula, B. Lucier, and I. Menache. Dynamic pricing and traffic engineering for timely inter-datacenter transfers. In *ACM SIGCOMM*, pages 73–86, 2016.

[11] S. Kandula, I. Menache, R. Schwartz, and S. R. Babbula. Calendaring for wide area networks. In *ACM SIGCOMM*, pages 515–526, 2014.

[12] A. Kumar, S. Jain, U. Naik, A. Raghuraman, N. Kasinadhuni, E. C. Zermeno, C. S. Gunn, J. Ai, B. Carlin, M. Amarandei-Stavila, M. Robin, A. Siganporia, S. Stuart, and A. Vahdat. BwE: Flexible, hierarchical bandwidth allocation for WAN distributed computing. In *ACM SIGCOMM*, page 1–14, 2015.

[13] N. Laoutaris, M. Sirivianos, X. Yang, and P. Rodriguez. Inter-datacenter bulk transfers with netstitcher. In *ACM SIGCOMM*, page 74–85, 2011.

[14] K. Nagaraj, D. Bharadia, H. Mao, S. Chinchali, M. Alizadeh, and S. Katti. Numfabric: Fast and flexible bandwidth allocation in datacenters. In *ACM SIGCOMM*, page 188–201, 2016.

[15] D. Salinas, V. Flunkert, J. Gasthaus, and T. Januschowski. Deepar: Probabilistic forecasting with autoregressive recurrent networks. *International Journal of Forecasting*, 36(3):1181–1191, 2020.

[16] A. Shaikh, C. Isett, A. Greenberg, M. Roughan, and J. Gottlieb. A case study of OSPF behavior in a large enterprise network. In *ACM SIGCOMM Workshop on Internet Measurment*, page 217–230, 2002.

[17] L. Song, J. Zhang, and B. Mukherjee. Dynamic provisioning with availability guarantee for differentiated services in survivable mesh networks. *IEEE Journal on Selected Areas in Communications*, 25(3):35–43, 2007.

[18] M. Suchara, D. Xu, R. Doverspike, D. Johnson, and J. Rexford. Network architecture for joint failure recovery and traffic engineering. *SIGMETRICS Perform. Eval. Rev.*, 39(1):97–108, June 2011.

[19] S. Taylor and B. Letham. Forecasting at scale. PeerJ Preprints 5:e3190v2 https://doi.org/10.7287/peerj.preprints.3190v2, 2020. [Accessed March 10, 2021].

[20] D. Turner, K. Levchenko, A. C. Snoeren, and S. Savage. California fault lines: Understanding the causes and impact of network failures. In *ACM SIGCOMM*, 2010.

[21] Yu Liu, D. Tipper, and P. Siripongwutikorn. Approximating optimal spare capacity allocation by successive survivable routing. In *IEEE INFOCOM 2001*, pages 699–708 vol.2, 2001.

[22] H. Zhang, K. Chen, W. Bai, D. Han, C. Tian, H. Wang, H. Guan, and M. Zhang. Guaranteeing deadlines for inter-data center transfers. *IEEE/ACM Transactions on Networking*, 25(1):579–595, 2016.
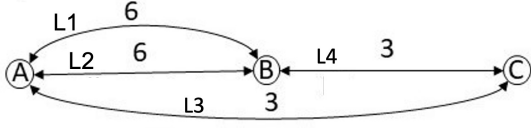
Figure 8: Toy Example

A Appendix

A.1 LP formulation

Objective Function: Minimize weighted sum of capacity augments where weight could be link cost, latency, etc.

$$\text{Minimize} \sum_l w_l \cdot X_l \qquad (1)$$

Subject to the constraints:

A. Demand constraints at sources and destinations:

$$\sum_{t_i \leq t \leq t_i + d_i} \sum_{l \in OL_{S_i} \cap L_{R_i}} v_{t,l}^{f,i} = V_i \qquad \forall f \in F, i \in D \qquad (2)$$

$$\sum_{t_i \leq t \leq t_i + d_i} \sum_{l \in IL_{T_i} \cap L_{R_i}} v_{t,l}^{f,i} = V_i \qquad \forall f \in F, i \in D \qquad (3)$$

Constraint (2) ensures that for any deferrable demand, i, the total volume of traffic emerging from its source node, S_i, over all time steps within its deadline is equal to the actual demand volume V_i. Constraint (3), on the other hand, ensures that the total volume traffic arriving at the destination node over all time steps within its deadline is also equal to actual demand volume V_i. In computing the total volume in constraint (2), we only consider the subset of links emanating from the source, S_i, that also lie on a valid path between S_i and the destination, T_i, and likewise in constraint (3).

B. Network flow constraint at each node, n:

$$\sum_{i \in D} \sum_{l \in IL_n} v_{t,l}^{f,i} + \sum_{\forall i | S_i = n} V_i = \sum_{i \in D} \sum_{l \in OL_n} v_{t,l}^{f,i} + \sum_{\forall i | T_i = n} V_i$$
$$\forall f \in F, n \in N, t \in T \qquad (4)$$

This is a network flow constraint to ensure that, in each time slice, the sum of the volume coming into the node and volume generated at that node is equal to the sum of volume going out of that node and volume sinking into that node.

C. Capacity constraint at each link, l:

$$(E_l + X_l) \cdot u_{t,l} \geq I_{t,l}^f + \sum_{i \in D} v_{t,l}^{f,i} \qquad \forall l \in L, f \in F, t \in T \quad (5)$$

This constraint ensures that the total volume of traffic that a link is able to carry during a time step (computed as a product of the total capacity provisioned on that link to accommodate the immediate and deferrable demands and the uptime of the link) should be greater than or equal to the sum of immediate and deferrable demands volume routed via that link during that time step.

Note that all of the above constraints apply during each failure scenario, f, so there is a set of such constraints corresponding to each such failure scenario.

A.2 Example of Cooperative Provisioning

We use a toy example network (Figure 8) with 3 datacenters and 2 immediate and 2 deferrable demands to illustrate steps 1 and 2 from Section 3.3. Table 4 shows the TTR and TBF of each link. For fair comparison with the baseline, we simulate 2-link simultaneous failures, in which links L1 and L3, and links L2 and L3 can fail together (the high TTR of 15 days for link L3 makes it more likely that its failures would overlap with those of the other links), along with all single link failures. Table 5 shows the input in terms of the immediate and deferrable traffic demands. Immediate demands R1 and R2 require a peak rate of 3 GB/day but sends 2 GB/day on average, hence sending total 60 GB in a period of 30 days.

1. Baseline :
Table 6 shows point-in-time failure scenarios simulated as part of the baseline provisioning described in the Section 4. Consider the point-in-time failure scenario 2 in Table 6, where links L1 and L3 fail together, hence the required rate between A and B is 6 GB/day to satisfy demands R1 and R3 and required rate between A and C is 9 GB/day to satisfy demands R2 and R4. And as L2 and L4 constitute the only available path, we need to allocate at least 15 GB/day capacity on L2 and at least 9 GB/day capacity on L4, hence capacity augments of 9 GB/day and 6 GB/day are required on L2 and L4, respectively. Provisioning for all such failure scenarios results in a total augment of 30 GB/day. With all demands expressed as peak rates that must be satisfied even when there are failures, point-in-time failure scenarios are effectively equivalent to a complete failure of a link over the entire 30-day period . Such a conservative approach is unnecessary for deferrable demands provisioning, as discussed next.

2. Approv:
Step I : We do peak-rate based point-in-time failure scenario provisioning for immediate demands which is similar to baseline. Considering the same point-in-time failure scenarios as in the baseline provisioning, we find that the topology in the

Failure Scenario	X1	X2	X3	X4
No failure	0	0	3	0
L1 and L3 fails	0	9	0	6
L2 and L3 fails	9	0	0	6
L4 fails	0	0	3	0
Final Augments	9	9	6	6

Table 6: Baseline provisioning with point-in-time failure scenarios

Failure Scenario	X1	X2	X3	X4
No failure	0	0	3	2
L1: [15, 17], L3:[15, 30]	0	0	0	5
L2: [15, 20], L3:[15,30]	0	0	0	5
L4: [15, 20]	0	0	5	0
Final Augments	0	0	1.67	5

Table 7: Volume based LP provisioning with timeseries failure scenarios

Link	TTR	TBF
L1	2	30
L2	5	60
L3	15	30
L4	5	60

Table 4: Link TTRs

Figure 8 is enough to satisfy the immediate demands, R1 and R2.

Step II: In this step, we provision for deferrable demands R3 and R4 on top of the Step 1 topology, while working with time series failure scenarios, as discussed in Section 3.4. Note that we have only shown a few failure scenarios (which cause the highest augments) due to lack of space but the final augments are based on simulating plenty of such failure scenarios.

Consider the failure scenario 2 in which L1 fails at day 15 and comes back up at day 17 and L3 fails at day 15 and comes back up at day 30. We don't need to provision extra capacity for demand R3 because if we send more volume of R3 between day 0 and day 15, we would not need to send any traffic between day 15 and day 17. On the other hand, since the demand R4 starts at day 15 and we need to send 90 GB in a period of 15 days, which coincides with the downtime of link L3, we would need to provision capacity on an alternate path to accommodate demand R4 in the face of such a link failure. Specifically, if we consider link L4 on the alternative path, 15 GB out of the total R4 demand of 90 GB could be accommodated in the valleys of immediate demand R2. That would leave 75 GB of R4 to be satisfied, necessitating 75/15 = 5 GB/day of extra capacity on link L4 to satisfy deferrable demand R4, alongside immediate demand R2 flowing over the same link, L4. Note that since we solve one LP optimally across all failure scenarios, the final capacity augments yielded by our LP is not necessarily the same as the maximum capacity augment required on each link across the individual failure scenarios.

Request	Type	Peak Rate	Start Time	Deadline	Volume
R1: A->B	Immediate	3 GB/day	-	-	60 GB (30 days)
R2: A->C	Immediate	3 GB/day	-	-	60 GB (30 days)
R3: A->B	Deferrable	-	0th day	30 days	90 GB
R4: A->C	Deferrable	-	15th day	15 days	90 GB

Table 5: Immediate and Deferrable demands

Some observations on Approv provisioning:

1. Baseline provisioning requires 30 GB/day total capacity augments whereas Approv requires only 6.67 GB/day total capacity augments.

2. For temporally overlapping demands, Approv will find optimal allocation. For example, deferrable demand R3 will send more bytes in the first 15 days to utilize the network well and will leave enough capacity on links L1 and L2 for sending R4 traffic between day 15 and day 30.

3. In our example, deferrable demands could be routed via either link L3 or link L4, but Approv returns higher augments on link L4 because it has lower TTR value. In effect, Approv favors augmenting low TTR and high TBF links, i.e., high-availability links.

4. As our objective function is to minimize total weighted capacity augments, Approv favors augmenting shorter paths.

5. With our volume-based LP formulation, we do not need to explicitly smooth our input demands, since Approv fills deferrable traffic in the immediate traffic valleys implicitly as part of its optimization.

OrbWeaver: Using IDLE Cycles in Programmable Networks for Opportunistic Coordination

Liangcheng Yu
University of Pennsylvania
leoyu@seas.upenn.edu

John Sonchack
Princeton University
jsonch@princeton.edu

Vincent Liu
University of Pennsylvania
liuv@seas.upenn.edu

Abstract

Network architects are frequently presented with a tradeoff: either (a) introduce a new or improved control-/management-plane application that boosts overall performance, or (b) use the bandwidth it would have occupied to deliver user traffic.

In this paper, we present OrbWeaver, a framework that can exploit unused network bandwidth for in-network coordination. Using real hardware, we demonstrate that OrbWeaver can harvest this bandwidth (1) with little-to-no impact on the bandwidth/latency of user packets and (2) while providing guarantees on the interarrival time of the injected traffic. Through an exploration of three example use cases, we show that this opportunistic coordination abstraction is sufficient to approximate recently proposed systems without any of their associated bandwidth overheads.

1 Introduction

The purpose of a computer network is to transmit messages to and from connected devices. The bulk of these messages are sent between two or more end hosts and are intended for use in applications therein (video streaming, web browsing, ssh terminals, stock trackers, etc). It is important to note, however, that networks are also frequently used for other purposes that are not directly related to end-to-end application traffic. These uses include but are not limited to control messages, keepalives, and probes.

In some cases, this second category of messages is sent over dedicated networks (e.g., an out-of-band control plane). Nevertheless, a significant portion is not, and for good reason. Multiplexing the traffic over a unified network results in more efficient resource utilization and helpful fate-sharing properties. For many uses, it is also required for correctness. For instance, active probing generally relies on the probe facing the same network conditions as normal traffic.

For in-band coordination, there is often a choice between fidelity and overhead. More so as many protocols use high-priority messages that directly cut into network capacity. For example, when deciding on an appropriate interval for sending routing-protocol keepalive messages, sending keepalives more frequently results in faster failure detection but at the cost of many extra packets in the network. Similarly, while techniques like congestion tagging [3, 22] and in-band network telemetry [27] can provide timely information about the recent state of network paths, they require either extra probe packets or space in the headers of existing packets, both of which occupy valuable bandwidth.

Given this tradeoff between fidelity and overhead, today's networks end up settling for a little bit of both. In some cases, the sacrifices are modest; in others, network operators are forced to limit the aggressiveness of their systems despite evidence of the benefits of finer granularity [6, 49]. In this paper, we argue that for a diverse set of protocols, the sacrifice is entirely unnecessary—systems can coordinate at high-fidelity with a near-zero cost to usable bandwidth and latency. In short: we can have our cake and eat it too.

Our system, OrbWeaver, is a framework for the opportunistic transmission of data across today's programmable networks. OrbWeaver takes advantage of gaps between user traffic and 'weaves' (i.e., injects) into *every* such gap customizable IDLE packets that convey information across devices. For modern, high-speed networks, these opportunities are plentiful. Crucially, OrbWeaver provides guarantees about the 'weaved' stream—guarantees on the maximum time between any two packets and guarantees on the impact of the injected packets on user traffic, switch resources, and power draw. A consequence of this predictability is that, even when there is no opportunity to send, the absence of IDLE packets reveals concrete information about the state of the network.

We note that a similar abstraction already exists at the data-link layer. In particular, in today's full-duplex Ethernet standards, the Physical Coding Sublayer (PCS) will fill any gaps in transmission with IDLE symbols [32, 41]. The continuous stream of incoming signals allows receivers to—with no impact to user traffic—test for corruption and link integrity at a fine granularity, even when there is no traffic on the network. Further, by continuing to transmit IDLE symbols after a link integrity issue has been raised, switches can also determine when the link becomes usable again.

OrbWeaver extends this technique to higher layers of the network stack by exploiting the data plane programmability, architecture, and packet generation capabilities of emerging programmable switching platforms. The resulting stream of packets can be used to generalize Ethernet's robust failure detection properties to a broader class of faults; however, its benefits go far beyond L3 failure detection. Rather, we demonstrate in this paper that with proper application, the nearly free communication that IDLE packets provide can be used to eliminate the fidelity-utilization tradeoff of solutions

to several classic problems in networking including clock synchronization and load balancing.

Implementing OrbWeaver's packet weaving presented several technical challenges. First, while IDLE symbols are part of the Ethernet standard and enjoy direct hardware/protocol support, to utilize today's devices and maintain their current performance, OrbWeaver must provide similar behavior without changes to switch architectures. Second, while many systems can benefit directly from opportunistic data transmission, many must continue to operate during periods where user traffic is already occupying all available bandwidth. To address the first challenge, OrbWeaver introduces a co-design of the selective data-plane filtering mechanisms and the rich priority configurations found in modern switches to guarantee minimal impact on user traffic. We verify the approach through a detailed examination of the specifications of the queuing subsystems on a Tofino switch along with experiments that stress-test worst-case behavior. To address the second, we introduce novel mechanisms that exploit IDLE packets and the guarantees of weaved streams to eliminate the bandwidth overheads of existing network protocols. We demonstrate these mechanisms through three case studies.

Our implementation[1] and evaluation demonstrate the efficiency and efficacy of OrbWeaver using real hardware, optical attenuators, and power meters. We find that, despite the introduction of the IDLE stream, OrbWeaver incurs negligible impact on user traffic, the computational/state resources of participating switches, or their power draw. We further demonstrate that this messaging substrate can be used to (re-)design recently proposed systems to eliminate their bandwidth overheads while closely approximating their performance.

2 Motivating Weaved Streams

This section presents the definition of a 'weaved' stream, its motive, and where data plane programmability can help.

Definition. A *weaved stream* is a union of user and IDLE packets traversing a link between two arbitrary network devices that provides two guarantees:

R1 That no link stays unutilized for too long. More precisely, there is some period τ where the interval between any two consecutive packets, d, satisfies $d \leq \tau$.

R2 That the injected packets do not decrease the effective throughput of user traffic or increase their loss rate.

Why weaved streams? Network protocols are, fundamentally, distributed computations that require coordination between different devices—sometimes adjacent devices, sometimes remote devices—for monitoring, control, and management. A perennial problem is how much bandwidth to allocate to these protocols, as each byte devoted to coordination is a byte that could have been used for user traffic instead. This tradeoff has tangible effects for many networking tasks:

[1]Code is available at https://github.com/eniac/OrbWeaver

- *Failure handling:* A common strategy for detecting the failure of remote network devices is the use of continuous keepalive messages. Here, each node periodically sends a keepalive to each of its neighbors; if a neighbor ever stops sending keepalives, nodes assume that they have failed. Fundamentally, the period between keepalives bounds the speed at which we can detect failures. Unfortunately, because keepalives are most accurate when sent over the same or higher-priority channels as user traffic, their sending rate is typically kept low (e.g., at a period of O(100 ms)) at the cost of slower detection.

- *Clock synchronization:* Prior work has also noted the utility of synchronizing network devices [29], e.g., for coordinated network updates [36, 45] or real-time streams [13]. Clock synchronization protocols typically pass messages that periodically compute the drift between the clocks of participating machines. Constant changes to not only the relative clocks but also the relative clock *rates* mean that more frequent updates can provide more accurate synchronization (at the cost of additional packets in the network, typically configured at a high priority).

- *Congestion notification:* Finally, this tradeoff can be seen in the detection/communication of congestion and current load. ACKs (and their corresponding loss/RTTs) are a particularly common method for inferring the presence of congestion, e.g., in TCP NewReno. As others have noted [3, 26], however, there are also advantages to more explicit signaling of the current congestion and queue statistics. Unfortunately, while effective, these statistics typically occupy packet header space or introduce additional packets into the network.

Over the years, network architects have developed many workarounds. These include hardware changes [29, 32], co-opting unused fields in headers [3, 50], carefully balancing the tradeoff for a particular service-level expectation [7], or otherwise coming to terms with the cost of coordination. Outside factors can guide the above decisions, such as whether ACKs are already necessary (e.g., for reliability) or if extraneous fields can be eliminated. However, in this paper, we ask a more fundamental question: are these tradeoffs necessary?

To that end, OrbWeaver is a framework for implementing network coordination that does not interfere with user traffic. OrbWeaver's weaved streams are both opportunistic and highly predictable—consuming every inter-packet gap of sufficient size but no more. Not every protocol can be implemented solely using weaved streams (though many can benefit from it). Even so, we demonstrate that at least for the three use cases above, weaved streams are sufficient to approximate state-of-the-art systems while reducing their impact on user traffic to virtually zero.

Why are there gaps? Usable gaps between packets can occur for many reasons, the most basic being application-level patterns and TCP effects. Indeed, prior work [37, 49] and
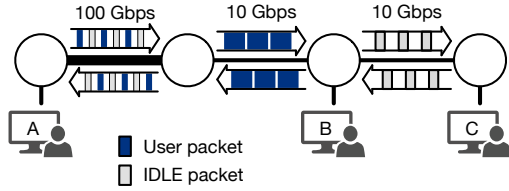
Figure 1: An example OrbWeaver-enabled network with four switches and three end hosts (connected with 10 Gbps links). A single two-sided connection $A \leftrightarrow B$ occupies the network, but a significant portion remains unused. Gaps between packets can occur for many reasons, but OrbWeaver can weave IDLE packets into all of those gaps.

our conversations with several large clouds/ISPs verify that micro-/milli-second inter-packet gaps are ubiquitous, even in networks that primarily handle large bulk-data transfers.

Gaps can also happen for structural reasons. For example, consider Figure 1 (sans IDLE packets). In it, a single connection $A \leftrightarrow B$ occupies all usable end-to-end bandwidth. Even if A and B pace packets perfectly, no host can send additional packets without displacing the existing user traffic, despite significant opportunities to do so (because of, e.g., congestion, link speed changes, and asymmetric connections). These gaps present a chance for opportunistic coordination.

Why now? OrbWeaver's ability to weave IDLE packets into gaps between user traffic is enabled by several features in modern switches: programmable data plane behavior, the capacity for local packet generation, and the ability to fully configure the queuing/prioritization of different traffic classes. We note that none of these are sufficient on their own.

For example, consider strict packet prioritization, which has been used for opportunistic bandwidth allocation [21, 24]. In SWAN [21], for instance, end hosts send low-priority background traffic to capture any bandwidth remaining after handling interactive and elastic services. A naïve application of these techniques, however, is a poor fit for in-network coordination, which occurs between devices in the network (as opposed to end hosts) and typically involves small data sizes that benefit from even short sending opportunities. Figure 1, for example, would not benefit from end-host actions.

3 Generating a Weaved Stream

Before we delve into the potential uses of weaved streams in Section 4, we first detail how to implement the abstraction in today's programmable switches.

Switch model. For simplicity, we primarily focus on the popular Tofino family of programmable networking devices (and discuss generalization to other types of devices in Appendix B). Figure 2 shows a conceptual diagram of the relevant components of the switches we consider. At a high level, when a packet enters from one of the Ethernet ports, its header is extracted by the programmable parser responsible
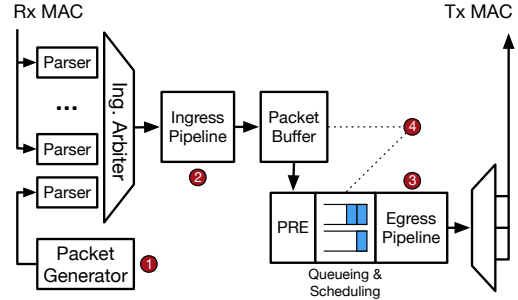
Figure 2: Conceptual diagram of the relevant components of an RMT switch, derived from the switch specifications in [12]. Only a single ingress/egress pipeline are shown. Circled numbers indicate steps and potential points of contention with user traffic that are handled in Section 3.1.

for that port. An ingress pipeline arbiter is then responsible for selecting one of the parsed packets and passing it through the ingress match-action pipeline.

After ingress processing, the packet will be placed in a shared packet buffer until it is ready to be sent out. Instead, the switch uses a shorter 'packet descriptor' for the next steps: optional replication by a Packet Replication Engine (PRE) (e.g., for multicast) and placement onto a per-port egress queue for eventual processing/deparsing. The data plane program and the traffic manager configuration decide whether an incoming packet should be buffered and whether a buffered packet should be enqueued for transmission.

Goal. R1 of the weaved stream abstraction requires a constant stream of packets on every link such that the union of user and IDLE packets satisfies $d \leq \tau$. We note that the optimal guarantee for τ is dependent on both the bandwidth, B, of each link and the MTU of the network. To see why, consider the extreme case where a user is occupying all of the bandwidth of a port i with MTU-sized packets. The receiver on the other side of the link will receive packets at a period of $\tau_i = \frac{\text{MTU}}{B_i}$, with OrbWeaver unable to inject any additional packets without impacting user traffic. Therefore, unless otherwise noted, OrbWeaver uses $\tau_i = \frac{\text{MTU}}{B_i}$ even if smaller IDLE packets would allow for faster injection.

In the worst case when there were zero user packets and N egress ports, the resulting target IDLE-injection rate is:

$$T = \sum_{i=1}^{N} \frac{B_i}{\text{MTU}}$$

For reference, for a 32 port switch with $B = 100$ Gbps and MTU = 1500 B, the per-port inter-packet gap, τ_i, is 120 ns, which results in $T = 266.7$M packets/sec.

Constraints. Complicating the injection of IDLE packets into the network are R2 and hardware constraints on the throughput of each switch pipeline, defined in terms of both byte-level bandwidth ($N \times B$) and packet-level bandwidth (proportional to the clock rate of the pipelines). For the latter, switches

typically provide guarantees up to a certain minimum packet size, and best-effort behavior for very small packets.

3.1 Mechanism Overview

OrbWeaver's IDLE-packet weaving leverages a combination of features found on our target platform: data-plane packet generation, data plane programmability, and fine-grained arbiter/scheduler configuration options. The switches' onboard per-pipeline packet generator modules, in particular, form a convenient substrate for our techniques. Using these modules, a network operator can create packets with predetermined content at a predetermined rate.

In principle, one could configure the generators to create packets at a rate T (thus providing OrbWeaver with its consistent stream of packets to convert into IDLE packets). Unfortunately, in practice, these generators do not have nearly enough capacity to satisfy the requirements of OrbWeaver. Moreover, blind injection of packets may interfere with the throughput, latency, or loss of user traffic. Instead, OrbWeaver uses the selective amplification method described below.

❶ **Packet generation.** The IDLE stream generation of Orb-Weaver begins with a low-rate but predictable stream of generated IDLE packets. The focus of this process is to provide a 'seed' stream with an emphasis on regularity; amplification up to T occurs later in the pipeline. More specifically, the generator module is configured to send a packet every $\frac{\tau_{\min}}{2}$ secs, where $\tau_{\min}$ is the minimum τ_i of any port on the pipeline.

There are two important aspects of this seed stream. The first is that the rate is double that of $\tau_{\min}$ in order to provide a degree of oversampling for the subsequent optimizations without sacrificing guarantees on the eventual spacing of packets. The second is that the IDLE packets are configured with a strict high priority at the ingress arbiter so that the packet will always be serviced as soon as it is generated. While this implies that IDLE packets are preferred over user traffic in the ingress pipeline, the low rate of this seed stream means that OrbWeaver incurs $<1.5\%$ overhead even for the worst case of minimum-sized packets sent at $\tau_{100\,\text{Gbps}}$ (denoting the optimal τ_i for a 100 Gbps link). More typical packet sizes and utilization eliminate the overhead.

❷ **Amplifying the stream on-demand.** OrbWeaver takes the low-rate seed stream above and amplifies it, potentially up to the full rate T, by leveraging another hardware feature found in modern switches: flexible multicast. In Figure 2, this behavior is implemented in the PRE, which can replicate a packet descriptor to the egress queues at line rate.

Unfortunately, the naïve approach of replicating a packet to every egress queue every $\tau_{\min}$ seconds can crowd out normal multicast packets and waste significant egress capacity. More specifically, there are two instances where it is not necessary to multicast a packet to a particular port i:

1. If the port is slower than the maximum speed, then sending at $\tau_{\min}$ will be too fast by a factor of $\frac{B_{\max}}{B_i}$.

2. If a user packet was already sent to the egress port recently, sending an IDLE packet is unnecessary.

OrbWeaver addresses both cases by oversampling the sending history of each port (at rate $\frac{\tau_{\min}}{2}$) and then selectively filtering/multicasting toward only the ports that need an IDLE packet. When a port is has bandwidth $B_i < B_{\max}$, the switch downsamples the IDLE packets by configuring two multicast groups (one with port i and one without) and picking the one with i every $\lceil \frac{B_{\max}}{B_i} \rceil$ packets. Similarly, if a port has sent a packet (user or IDLE) in the past $\frac{\tau_{\min}}{2}$ seconds, we can select a multicast group that does not contain the given port.

Concretely, this filtering step uses a single stateful register entry with a bit width equal to the number of ports attached to the pipeline. In essence, the register is a bitvector where each bit represents whether we have sent a packet to the associated port within the last $\frac{\tau_i}{2}$ seconds. For every incoming seed packet, if the associated bit is 1, we omit the port and flip the bit to 0; if the bit is originally 0, include the port in the multicast and flip the bit to 1. Specifically:

```
user packet: filter_reg |= 1 << egress_port
seed packet: filter_reg ^= speed_mask
```

When all ports are the same speed, `speed_mask` is always $2^N - 1$; for hybrid configurations, the ith bit is 1 for every $\lceil \frac{B_{\max}}{B_i} \rceil$ packets and 0 otherwise. After updating the register, OrbWeaver multicasts the current seed packet to the multicast group specified by `filter_reg` (in particular, its value before the xor)—if and only if bit i in the multicast group ID is 0, port i is included in the multicast.

In principle, a direct application of the above filtering step guarantees that the PRE will have enough bandwidth for all user multicasts, assuming that each user multicast results in at most one packet on each egress port. Two aspects of modern switch design potentially complicate this design.

The first is that today's switches typically cannot support a unique multicast group for each of the 2^N possible combinations of target ports. OrbWeaver addresses this by reducing the number of groups by coalescing ports into groups of M such that, if any port in the set has its bit in `filter_reg` set, the entire set receives the multicasted packet. This approach trades a factor of 2^M reduction in the number of multicast groups for a worst-case $\frac{M-1}{N}$-factor decrease in PRE bandwidth. The second is that modern switches are often composed of different pipelines, each supporting distinct packet generators, sets of registers, and groups of ports. Lack of visibility across pipelines means that `filter_reg` may only track local sends, which can also lead to higher PRE usage.

We note, however, that in both of the above cases, Orb-Weaver will only incur false negatives (and no false positives) of user packet presence, thus satisfying R1. We also note that very few modern networks are continuously multicasting to all ports at near line-rate.

❸ **Weaving the IDLE stream between user packets.** After the stream is amplified, it reaches the egress queues and

pipeline of the switch. To bound the impact of the stream on user traffic, OrbWeaver configures its packets to have a strictly lower priority than any other user traffic on the same port. If there is user traffic to send, the IDLE packets will not impact them; if there is no traffic to send, the IDLE packets will be sent at a minimum rate of τ_i per port i. The only potential impact to the latency/throughput of user traffic is when an IDLE packet is scheduled just before a user packet arrives, in which case the user packet will be delayed by at most $\texttt{pkt\_size}/B_i$. The delay is only incurred once per packet burst, which implies a bound on OrbWeaver's end-to-end impact on latency and throughput.

Upon arriving at the ingress pipeline of the downstream switch, the packets will be dropped. This also has near-zero impact on user traffic as IDLE packets are only received when the upstream switch has nothing to send.

❹ **Managing the packet buffer and egress queues.** Finally, through the above process, there are two primary places where IDLE packets can compete with user packets for memory in addition to bandwidth. The first is the per-egress output queues that hold packet descriptors before they are serviced by the egress pipeline. The second is the shared packet buffer that stores packet contents until they are sent out on the wire.

To bound the impact of OrbWeaver on both resources, we statically carve the buffer using egress and ingress buffer accounting mechanisms, respectively. For the former, we note that the queue for IDLE packets (the lowest priority queue for the port) is distinct from those of user packets. This queue only needs to be one cell deep as another IDLE packet is guaranteed to arrive in a timely fashion, and thus, the impact on aggregate memory capacity is negligible. For the latter, we can likewise keep the required buffer shallow because of the guarantees of the packet generation process. Specifically, we can confine the IDLE packets to a fixed-size, non-shared region of the packet buffer. The buffer only needs to have a depth equal to the sum of the egress, per-port IDLE-packet queues plus a small amount of headroom for any potential cycle-level processing delays. This is $< 0.01\%$ of the total buffer size of a typical modern switch.

3.2 Evaluating the Weaved Stream

In this section, we delve deeper into OrbWeaver's potential impacts on user traffic. We do this with the assistance of a prototype implementation on a $2\times$ Wedge 100BF-32X testbed. Additional experiments can be found in Appendix F.

3.2.1 Can OrbWeaver Inject at Rate T?

To demonstrate that our approach can achieve T on a fully provisioned switch, we validate it empirically. Specifically, we configure a switch with all 32 ports active and running at a full 100 Gbps. We then configured the switch's packet generator module to generate seed packets at a rate of $2/\tau_{100\,\text{Gbps}}$ and then multicast every other IDLE packet to all ports.
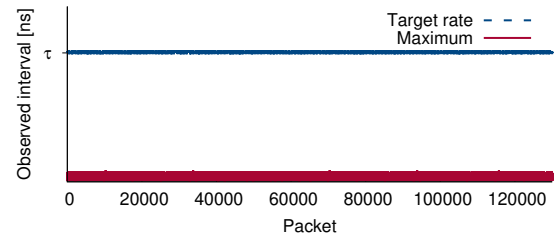
Figure 3: An empirical evaluation of the switch's capacity to generate IDLE packets. Packets were injected to all ports, but the graph depicts the observed inter-packet gap at only one of those ports. Results are shown for both the target rate ($B_i = 100\,\text{Gbps}$, MTU $= 1500\,\text{B}$) and the maximum achievable rate. y-axis omitted to protect confidential information.

Figure 3 shows a time series of the interval between IDLE packets, as observed by the egress pipeline of a single port. To record the series, we maintained a ring buffer (implemented via a data plane register) of the difference between the current $\texttt{egress\_global\_tstamp}$ and the previous. The observations were maintained in the egress pipeline and for a single port (other ports' results are identical).

We find that, not only is the injected stream able to achieve $\tau_{100\,\text{Gbps}}$ for every port simultaneously, the observed rate is stable across packets. Further, increasing the amplification factor of the multicast configuration enables IDLE packet generation more than an order of magnitude faster than the target interval, $\tau_{100\,\text{Gbps}}$. Among other implications, this means IDLE packet injection is robust to higher bandwidth and lower MTUs, even without improvements to packet replication capacity.

3.2.2 Can OrbWeaver Bound Packet Gaps?

In addition to being able to generate IDLE packets at rate T, R1 also requires regularity in the form of a bound on the gap between packets. We note that Figure 3 already demonstrates the regularity of this gap on a switch without traffic. We also note that in the other extreme (when ports are always congested), R1 is trivially satisfied.

In this section, we extend these results to a network with burstiness and varying levels of traffic. Specifically, we use a hardware testbed consisting of two OrbWeaver-enabled switches (A and B) and a set of servers connected to A. User traffic is passed hosts$\rightarrow A \rightarrow B$ with amplification to fully utilize the ports at B. For this experiment, we used $\texttt{tcpreplay}$ and pcap traces from an ISP backbone [9] and a cloud data center [8]. We set up a register in the ingress pipeline of the downstream switch B to record the distribution of the interval between consecutive packets.

Figure 4 shows the results for a single 25/100 Gbps port. Without OrbWeaver, very few intervals are under τ for the target link speed, and the tail is very long. OrbWeaver, on the other hand, is able to weave in IDLE packets to guarantee an upper bound on the packet interval regardless of the original traffic pattern. In particular, for a configured generation interval of t ns, out of 2.14×10^9 interarrival periods, the max-

(a) 25 Gbps ISP backbone

(b) 25 Gbps data center
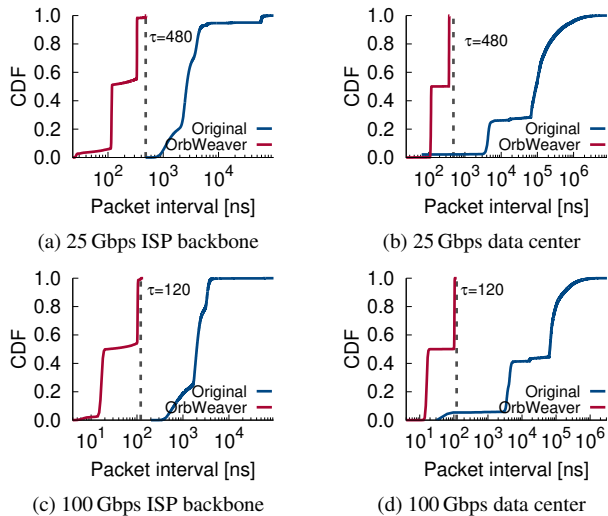
(c) 100 Gbps ISP backbone

(d) 100 Gbps data center

Figure 4: Observed intervals between packets with/without OrbWeaver's weaved stream. The dotted line shows the ideal period τ for each link speed. Without OrbWeaver, the maximum interval was >100s of ms but we truncate for readability.

imum observed interval was $(t+3)$ ns (observed for only 32 intervals). The discrepancy is likely due to either clock drift or the aforementioned cycle-level processing delays. Notably, the presence or absence of cross traffic had negligible effect on the frequency of these 3 ns outliers so in practice, we can set $t = \tau - 3$ and achieve reliable results.

Explanation. The regularity of OrbWeaver's weaved stream derives from the architecture of the switch and the mechanisms of OrbWeaver. From the components of Figure 2, the parser used by the packet generator is separate from those of the external traffic, the ingress pipeline grants strictly higher priority to the generated packets over external traffic (user or IDLE), and the packet buffer protects IDLE packets from interference through static reservations for worst-case capacity. When combined, a generated IDLE packet can only be delayed through HoL blocking when an external packet arrives just before the generated packet. For unicast packets, this is a 1-cycle delay; for full broadcasts, this is up to an N-cycle delay (which is short for today's high-speed networks).

At the egress pipeline, the priorities are reversed: IDLE packets are set to a strictly *lower* priority than user traffic. This change stems from a change in objective: in the egress pipeline, it is no longer necessary for the IDLE packets to be sent at a precise rate; instead, the goal is to send *any* packet at above the minimum rate, τ_i. Choosing a user packet instead of an IDLE one can only decrease the inter-packet gap.

Note that, in a Tofino, these priorities (unlike those at the ingress) are only effective within their respective ports. Thus, the switch will send a low-priority packet on port i even if there is a higher-priority packet queued for a different port. As long as the average packet size is above the minimum for line-rate processing, ports can be considered in isolation.
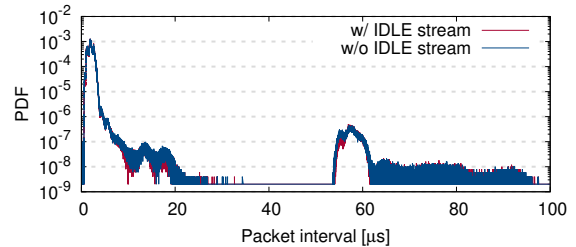
Figure 5: The impact of IDLE packets on user traffic at the ingress pipeline with/without a generation rate of $2/\tau_{100\,\text{Gbps}}$.

3.2.3 Do IDLE Packets Affect External Traffic?

As important as the impact of cross traffic on generated IDLE packets are the impacts of the generated packets on (1) user traffic and (2) incoming IDLE packets. A significant impact on (1) implies violations of R2; on (2), it implies inaccuracy in inter-arrival times and potential violations of R1. We discuss potential impacts in the two pipelines separately.

Ingress pipeline. While OrbWeaver's packet prioritization means that IDLE packets will be preferred over external traffic in the ingress pipeline, its use of multicast amplification reduces their impact to 1.5% of maximum packet-level capacity, with zero impact to byte-level capacity.

To evaluate the practical effects of this overhead, we replayed a real-world packet trace over an ingress pipeline of an OrbWeaver switch. The packet trace was generated using `tcpreplay` and link-level packet traces captured from 10 Gbps Internet routers [9]. To saturate the pipeline, we sped the traces up to match our setup's 100 Gbps per-link bandwidth and replicated them to fill the switch.

We compare two cases. In the first, only the above external traffic is present. In the second, we used the exact same traces but, in parallel, we injected IDLE packets into the same pipeline just as we did in the previous subsection. In both cases, we measured the packet count and interarrival times of user packets in the ingress pipeline with the help of stateful registers that aggregate both statistics.

We find that, for the speeds and packet sizes in the evaluated trace, the throughput and congestion loss of user traffic is the same whether the generated IDLE stream is present or not. The only metric that is impacted is latency, where a slight delay can be introduced each time a generated packet is processed one 'clock cycle' ahead of a user packet; however, this is minor and mitigated by the low frequency of IDLE packet injection. Figure 5 depicts the cumulative impact of this delay using a histogram of the packet interarrival time of the traces, with and without the IDLE stream—the majority of the differences are due to randomness in `tcpreplay` between executions, rather than OrbWeaver.

Egress pipeline. The benefits of the amplification strategy to contention mitigation stop at the PRE, but two other factors take its place in ensuring that user traffic is not impacted in the egress pipeline. The first factor is the filtering step that was introduced in Section 3.1, which prevents superfluous
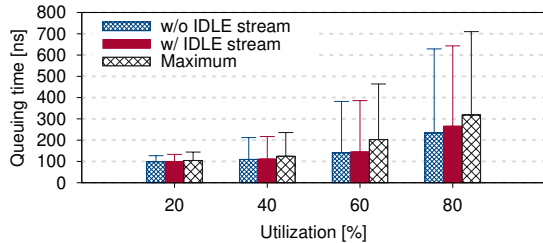
Figure 6: The impact of IDLE packets on the latency of user traffic at the egress pipeline. Results are shown for various levels of average utilization. 0% and 100% are not shown as OrbWeaver becomes trivially optimal. To provide an upper bound on the impact, we disable adaptive ingress filtering and populate the pipeline with only small (64 B) user packets. A real OrbWeaver deployment would have much lower impact.
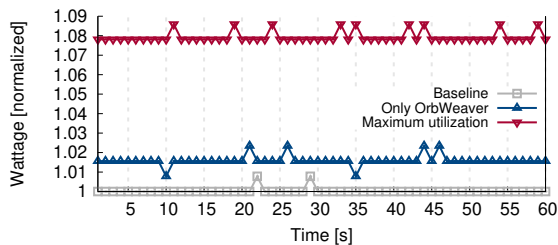

Figure 7: The power draw of a OrbWeaver switch compared to that of an idle (baseline) and a maximally utilized switch. Y-axis is normalized to the average power draw of the baseline.

usage of both the PRE and egress pipeline when the egress ports are already occupied. For IDLE packets that are not filtered in the ingress pipeline, the second factor is the strict prioritization of user traffic over IDLE packets of the same port, also introduced in Section 3.1. The second factor, in particular, provides an upper bound on the impact of the IDLE packets as long as the user traffic respects the minimum average frame size requirements of the switch specification (see Appendix D for a formal analysis).

To truly stress these mechanisms, we evaluate an extreme scenario in which multiple hosts send minimum-size (64 B) packets toward a single egress port and OrbWeaver's filtering mechanism is disabled. This situation is not possible in OrbWeaver, but is helpful in demonstrating the efficacy of egress prioritization for protecting user traffic. The results verify the analysis above, even for high user-traffic utilization. For comparison, we also show the impact of an IDLE stream operating at the order-of-magnitude-higher maximum rate of Figure 3 but still set to low-priority. Again, across all experiments, throughput was unaffected.

3.2.4 Does Injection Affect Power Usage?

Finally, we investigate the impact of weaving on the power consumption of today's switches. A natural concern is that the continuous stream of packets will increase consumption; however, we find the actual impact is minimal as the underlying Ethernet MAC already continuously sends IDLE symbols.

To evaluate this, we used a P3 Kill-A-Watt Electricity Us-
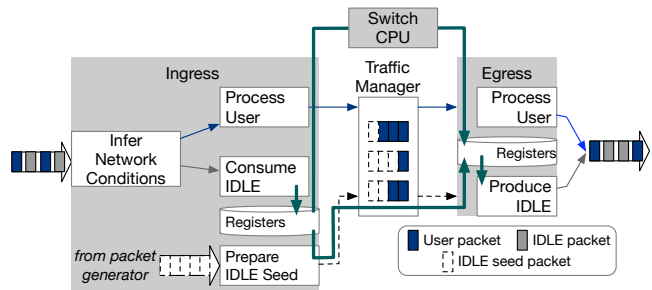
Figure 8: Structure of a P4 program that processes a weaved stream. The ingress pipeline extracts information from the weaved stream, then processes user and IDLE packets separately. The egress pipeline processes user packets and transforms seed packets into IDLE packets. Pipelines can communicate using registers that are synchronized with either seed packets or the switch CPU, as shown by the thick lines.

age Monitor (Model P4400) to measure the total power draw of a Wedge100BF-32X programmable switch. The monitor sits between the switch's power plug and its power outlet and can measure wattage to within 0.2–2.0%. To emulate the switch's deployment into a network of programmable switches, we connect every port on the switch to a second switch that logically functions as 32 neighboring switches. We test three distinct configurations:

- *Baseline:* All ports on the switch are connected at 100 Gbps; however, the switch is otherwise inactive, i.e., there is no incoming traffic nor any IDLE packets.

- *Only OrbWeaver:* Same as above, but with OrbWeaver's IDLE stream generation enabled on all switches. The switch is, thus, both sending and receiving packets at T.

- *Maximum utilization:* The worst case scenario, where the switch is both sending and receiving user packets at the maximum rate and generating IDLE packets (that are eventually dropped in the ingress pipeline).

Figure 7 shows the power draw of each configuration over a 1 min period. OrbWeaver's transmission of packets at rate T increases the average power draw of the switch by <2%.

4 Use Cases

Figure 8 outlines the general structure of a P4 program that uses OrbWeaver. Whereas a standard P4 program processes a stream of user packets, an OrbWeaver P4 program processes a weaved stream of user and IDLE packets. OrbWeaver programs can append/read information from the payloads of the IDLE packets (which appear on the wire as a special Ether-Type) or infer statistics from the timing of the weaved stream. In either case, the content of IDLE packets can be manipulated just like any other packet (metadata like the drop decision, priority, or egress port should not be changed).

In typical usage, the receiving switch will process, record, and drop incoming IDLE packets before the end of the ingress

pipeline. In most cases, the IDLE packets bypass the normal pipeline logic and, thus, will not affect user byte/drop/error counters. Separately, they use either (*a*) an agent on the switch CPU [47] or (*b*) a locally generated IDLE seed packet to transfer data from the ingress to the egress pipeline before sending to the downstream switch. Together, they facilitate multi-hop communication over IDLE packets.

In this section, we detail three example use cases of Orb-Weaver (see Appendix A for others). For each example, we consider a recently proposed network system, and we explore how well OrbWeaver can approximate it without introducing any additional impact on user traffic. We note that, in some cases, this restriction can result in suboptimal designs (i.e., imposing on user traffic may result in better overall performance, even if it incurs overhead). Rather, we ask: how far can operators go before needing to ever consider the choice between network throughput and features?

4.1 Use Case #1: Fast Failure Detection

Failures of network components are common in large networks where the number of devices involved ensures a constant flow of incidents. Reasons for the failures include overheating components, power instability, bit flips in the signal, loose transceivers, bent fibers, or any number of other causes [15, 44, 51, 52]. In the end, however, the symptom of many of these failures is the same: lost packets in the network.

Thus, as the first steps toward mitigation, quickly detecting and quantifying packet loss is critical to maintaining high availability and stringent SLOs, particularly as networks improve in both bandwidth and reaction time such that control-plane processing is no longer the sole bottleneck [11, 26, 30–32, 47]. Unfortunately, as mentioned in Section 2, common detection approaches—periodic keepalives or pings—force network architects to sacrifice detection latency to constrain overheads. Moreso as pings are traditionally prioritized over user packets to minimize false positives.

Even recent systems like NetSeer [50] that track userpacket loss inband (without injecting additional packets) suffer from this tradeoff. For example, NetSeer's choice to not inject additional packets means that the network is necessarily slow to detect a black hole (differentiating from a lack of demand requires CPU coordination to compare the flow counters of adjacent switches). Likewise, their choice to tag every packet with a sequence number incurs a bandwidth overhead of 0.3%~6.3% in return for higher detection granularity (unless there are previously unused bits in the header and we cannot change the data plane to remove them).

4.1.1 An OrbWeaver Redesign

Taking NetSeer as a base, we can replace its inter-switch communication with an OrbWeaver-influenced design to eliminate bandwidth overheads and significantly improve detection time. We refer readers to the original paper [50] for full details of the existing system but summarize the relevant components

as follows. NetSeer records the 5-tuple of each packet in the egress pipeline using per-port ring buffers and tags it with a 4-byte sequence number. The downstream switch stores the last observed sequence number. Upon detecting a gap (e.g., packet 14 after packet 12), it sends 3 duplicate and highpriority drop notifications to the upstream switch for each missing sequence number. If the upstream switch receives at least one such notification, it will use the records in the ring buffer to generate a flow event for the missing packet, which will be compressed/summarized for the management plane.

In NetSeer-OW, switches maintain per-port hash tables that, like NetSeer, record the 5-tuples and packet counts of passing flows (using the 5-tuple hash as the index). The caches are maintained in the egress pipeline of each upstream switch as well as the ingress pipeline of each downstream switch. As channels are FIFO and the tables use the same size and deterministic hash function, their content should always be identical. The only exceptions occur after a packet loss, at which point either a counter or a 5-tuple will differ.

In this re-design, user packets are *not* tagged with any additional data nor does it require triple-notifications. Instead, the upstream switch will opportunistically embed in IDLE packets psuedo-randomly selected cache records[2]. If the downstream switch finds that a record differs from its local copy, it will generate an event for the contained 5-tuple. It will also generate an event if packets stop arriving, which is detected with locally generated IDLE seed packets that scan per-port weaved-stream counters. After NetSeer-OW compresses/filters these events, the control plane sends the results over a low-priority TCP connection to the central controller.

Note that, in addition to exploiting the IDLE stream to carry flow information, (R1)'s guarantee of packet arrival rates enables provably optimal detection speed of link failures. In principle, OrbWeaver can trigger an alert if the `ingress_mac_tstamp` of any two consecutive packets is $\leq \tau$. While that level of granularity may be too aggressive for many networks, we note that recent proposals for data plane rerouting have made detection speed a bottleneck [11, 26, 30, 32], particularly if a goal is zero-loss failure recovery. In the end, the point is that OrbWeaver can provide arbitrarily precise failure detection/statistics for current and future networks.

Dealing with a lack of sending opportunities. While extended periods of maximum utilization are rare in most networks [9, 38, 49], NetSeer-OW can still provide useful properties during these extreme conditions. For example, for failure detection, a downstream switch in a fully utilized network can immediately detect a packet drop by examining the gaps between adjacent packets (a drop occurred when the gap > τ).

Flow attribution is slightly more challenging, with the chief concern that the switch evicts the flow before including it in an IDLE packet. We can quantify the probability of this hap-

---

[2]To improve the update rate, we can pack up to three 5-tuple-counter records (IPv4 and counters of 3 B) in each packet. To handle register access limitations, we can pack the records or split the table across multiple arrays.

	NetSeer		**NetSeer-OW**	
Data structure size (per-port)	**256**	**64**	**512**	**128**
SRAM (KB)	384	192	896	320
Number of sALU/register arrays	6	6	7	7

Table 1: Data plane resource usage for typical NetSeer and NetSeer-OW configurations on a 64×100 Gbps switch.

pening using the formalization in Appendix E. For reference, using the assumptions of Appendix E, average utilization of [9, 38], and flow cache performance of [39], ISP routers with 128 cache entries per port would have a $P(\text{notified}) \approx \frac{0.72}{0.72+0.28*0.45/3} = 94.4\%$. A data center switch with 128 cache entries would have $P(\text{notified}) \approx \frac{0.75}{0.75+0.25*0.16/3} = 98.2\%$, or with 512 entries $P(\text{notified}) \approx \frac{0.75}{0.75+0.25*0.05/3} = 99.4\%$.

Benefits. Compared to the original NetSeer design, the primary benefit of the OrbWeaver augmentation is to completely eliminate *all* sources of bandwidth overhead—in essence, we can apply NetSeer for 'free.' In particular, it eliminates the overhead of sequence number tagging (0.3%~6.3%) of capacity; the replicated, high-priority failure notifications (up to 100% of reverse link capacity); and the impact on user traffic of the event reports. Beyond overhead, it also improves the speed to detect inter-switch failures, particularly during periods of low utilization.

Table 1 shows the data plane memory consumption of both systems. Additional memory increases $P(\text{notified})$, however the relationship is different for each system. As a concrete data point, consider the coverage goal highlighted in the original NetSeer evaluation [50]—to correlate 90% of packet loss events with flows. For a 64×100 Gbps switch and a similar estimation strategy as above, NetSeer-OW meets this goal with 320 KB of SRAM (128 cache slots per port) in both ISP and data center workloads. On the other hand, assuming the network's minimum packet size is 64 B, NetSeer requires approximately 384 KB of SRAM to meet the 90% coverage objective because it must allocate enough ring buffer slots per port (256) to ensure that sequence numbers are not overwritten before switches have a chance to correlate their results.

4.1.2 Evaluation

Detecting failures more quickly. To quantify how quickly NetSeer-OW can detect a failure, we deployed NetSeer-OW to a hardware testbed and randomly disconnected a link between the two switches *A* and *B* 100 times to emulate 100 fail-stop link failure events. To test the limits of our approach, we configured the probes to mark a τ-timeout failure as soon as even a single packet loss is detected.

Figure 9a shows the detection time of trials for 10, 25, and 100 Gbps links. NetSeer-OW achieved 100% precision and recall. It also consistently detected the failure within 10s of nanoseconds of the optimal time. In contrast, typical configurations for protocols like Bidirectional Forwarding

(a) Link fail-stop detection (b) Link fail-stop recovery
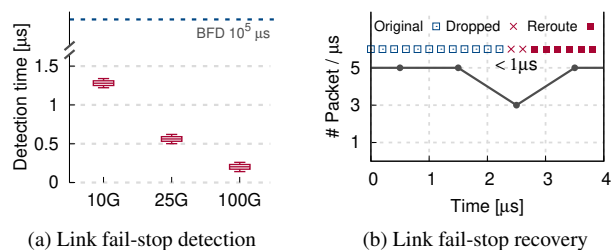
Figure 9: (a) the min, Q_1 (p25), median, Q_3 (p75), and max of OrbWeaver's time to detection across 100 failure events. (b) OrbWeaver's time to recovery (<1 μs) from a bidirectional failure of a 25 Gbps link. A total of two packets are lost.

(a) Completion time during failures (b) Optical attenuators
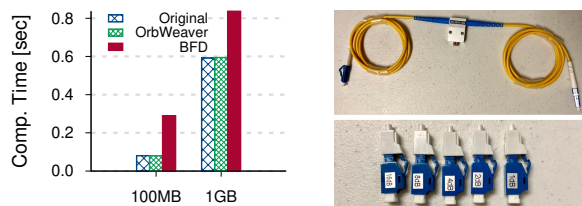
Figure 10: (a) shows the transfer completion time comparison for original, NetSeer-OW, and BFD (100 ms) in a simple leaf spine topology. With NetSeer-OW's fast detection and data plane reroutes, the impact is minimal.

Detection (BFD) are closer to 10s or 100s of milliseconds; even recent data plane detection systems [20, 30] are several orders of magnitude slower than NetSeer-OW can achieve.

Figure 9b shows the resulting seamless recovery when NetSeer-OW is combined with a simple data plane rerouting mechanism. In the experiment, we induce a bidirectional failure in one link between *A* and *B*, and we configure *B* to failover to a backup path as soon as it detects an error. On top of this setup, we send a steady stream of packets on the target link at a relatively high rate of 5M packets per second. A total of two packets were lost—likely in-flight.

End-to-end impact. To evaluate the end-to-end impact, we emulate a leaf-spine topology with 2 leaf switches L1, L2 and 2 spine switches S1, S2. All switches run OrbWeaver with pre-computed data plane backup paths. Between L1 and L2, we insert a variable fiber optic in-line attenuator capable of 0~60 dB attenuation. On hosts connected to the leaf switches, we run TCP transfers of varying sizes using iperf, during which we increase attenuation from zero until failure and examine the impact over the transfers experiencing the events. As Figure 10a shows, with OrbWeaver, the impact of failure is negligible with respect to completion time. In contrast, with BFD, failures cause the 100MB transfers to take over $4\times$ longer and the 1GB transfers to take over 30% longer.

4.2 Use Case #2: Time Synchronization

Time synchronization is another common task in modern networks. Like failure detection, time synchronization requires
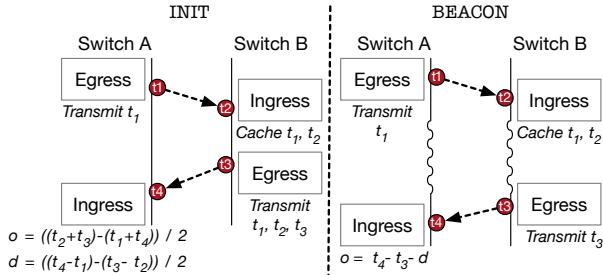
Figure 11: Time sync in DPTP-OW, using IDLE packets. When the difference between t_2 and t_3 is small, *A* treats the message as part of an `INIT` phase and calculates o, the clock offset, and d, the one way delay. When it is high, the `BEACON` phase uses the most recent d to track clock frequency drift.

coordination between adjacent switches, and many other applications rely on its accuracy [13, 36, 45, 46].

Unfortunately, the most common methods for synchronizing time between adjacent machines involve the computation of One-Way Delay (OWD) using periodic, high-priority echo requests/replies [1, 14, 30]. Here too, architects are presented with a tradeoff: clock frequency drifts imply that the faster we send echoes, the more closely we can bound the clock offset and the more accurate the synchronization. Protocols like DTP [29] that integrate the protocol into the physical layer can circumvent this overhead but require hardware changes.

4.2.1 An OrbWeaver Redesign

The state-of-the-art in time synchronization for programmable switches is DPTP [25]. In it, two adjacent switches (a client, *A*, and a server, *B*) compute the offset of their local clocks by leveraging switches' ability to embed timestamps into each packet during different stages of packet processing. Host and multi-hop synchronization are also possible using multiple strata. The protocol calls for three messages in each round of the protocol: (1) a DPTP request $[A \rightarrow B]$, (2) a DPTP response $[B \rightarrow A]$, and (3) a DPTP follow-up $[B \rightarrow A]$. All three messages are high-priority to eliminate queuing delay.

(1) is timestamped using the Tofino `egress_deparser_tstamp` and `ingress_mac_tstamp` of *A* (t_1) and *B* (t_2), respectively. (2) is timestamped using the same counters in *B* (t_3) and *A* (t_4), respectively. In a traditional clock synchronization protocol, the offset would be computed as $\frac{(t_2+t_3)-(t_1+t_4)}{2}$. Unfortunately, we note a fundamental limitation of today's programmable switches—that the `egress_deparser_tstamp` does not capture the actual point of packet serialization. Thus, the computed offset is subject to variable delays as a result of egress MAC contention. As a result, DPTP introduces the third packet, the follow-up, which embeds a more accurate egress serialization timestamp (obtained out-of-band). Again, we refer interested readers to [25] for full details.

An OrbWeaver-inspired redesign can obviate the need for the third, follow-up message by inferring the egress MAC contention from the weaved stream (and only using results

with no contention). This allows us to use the traditional two-way protocol of Figure 11. It can also eliminate the impact of the remaining messages using opportunistic sends.

Opportunistic synchronization: Rather than relying on high-priority echoes, a system can rely solely on OrbWeaver's IDLE packets to piggyback timestamps. In particular, whenever *A* has an opportunity, it sends a request to *B* on an IDLE packet with a field for t_1. Upon receiving the packet, *B* maintains a cache for the most recent values of t_1 and t_2. Separately, whenever *B* has an opportunity, it sends the most recent values of t_1 and t_2 along with the local `egress_deparser_tstamp` in t_3. In an empty network, *A* can calculate the clock skew as $\frac{(t_2+t_3)-(t_1+t_4)}{2}$ just as DPTP but with much more frequent synchronization (leading to lower *jitter*, i.e., nominal error [33]).

A challenge with the above approach occurs in networks with high utilization. The traditional OWD estimation method used above implicitly assumes that the clock drift is constant for the duration of the protocol round; otherwise, the delays at the time of the request and response may not be comparable due to clock frequency drift. In OrbWeaver, this can happen if there is congestion from *B* to *A*; the gap between t_2 and t_3 can be unbounded, leading to inaccurate results.

We address this challenge by borrowing an idea from a different protocol, DTP [29]: the decoupling of synchronization into INIT and BEACON rounds. If the time between t_2 and t_3 is sufficiently small, the round is treated as an INIT round and *A* computes the offset as above. Otherwise, *A* treats the message as part of a BEACON round where it takes d, the OWD computed from the last INIT round ($d = \frac{(t_4-t_1)-(t_3-t_2)}{2}$), and it computes a new offset: $o' = t_4' - t_3' - d$.

Selective synchronization: Finally, to remove the need for DPTP's third 'follow-up' message, we can exploit the implicit information contained in the woven stream's timing. The underlying intuition is simple: if the gap between an IDLE packet and its preceding packet is less than τ, the IDLE packet may have encountered contention at the egress MAC. In this case, the packet's timestamp may be unreliable. DPTP corrects for this contention with the follow-up message; Orb-Weaver simply ignores these protocol rounds. While this filtering effectively requires that usable gaps be $> \tau \sim 2\tau$, it greatly improves the accuracy of the protocol while still permitting frequent re-synchronization in modern networks.

Dealing with a lack of opportunity to send. The above protocol fully synchronizes switches when both links have concurrent IDLE gaps. The protocol also includes support for correcting small drifts when only one direction has a gap (by adjusting to the fastest clock in the network). We note that in a network with multiple paths, we can configure synchronization to propagate among any one of those paths. Thus, if we view the network as a directed graph, the only time a switch may lose synchronization is if sufficient links are maintaining 100% utilization that the links form a cut of the graph. In the end, if operators need assurances, they may need to send
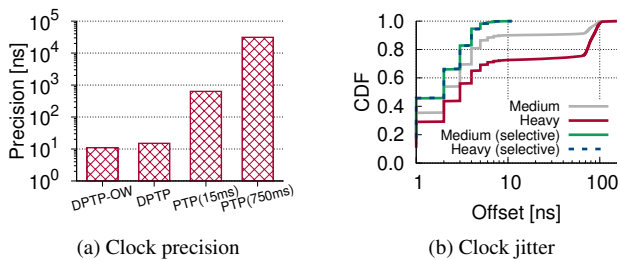
(a) Clock precision (b) Clock jitter

Figure 12: (a) shows the precision for different synchronization protocols and a heavy workload (~80% CAIDA user traffic). (b) shows the CDF of observed offsets (absolute value) for DPTP-OW upon medium and heavy loads for 10 Gbps link ($\tau = 1200$ ns), w/ or w/o selective sync. OrbWeaver achieves a precision of 11 ns even under heavy user traffic.

higher-priority messages if too much time elapses; however, we can extend our techniques so that the messages only need to be prioritized above the lowest-priority user traffic—high-priority, interactive applications would be unaffected.

Benefits. As long as there is occasional usable bandwidth in the network, OrbWeaver again eliminates all bandwidth overheads without sacrificing accuracy or nominal error. When the network is underutilized, it actually provides similar re-sync intervals as DTP but using commodity PISA switches.

4.2.2 Evaluation

Following prior work, we evaluate DPTP-OW's precision [29, 43] (defined as the maximum clock skew in the network), as well as its jitter [33] (defined as the distribution of measured offsets or nominal error). Again to match prior work, we evaluate these in a two-switch testbed during a 20 min collection for 10 Gbps link with a medium workload (a CAIDA trace with 25% average utilization) and a heavy workload (the same trace sped up to ~80% average utilization). We compare to both DPTP (with 2000 requests/sec) and PTP. For PTP, prior work has suggested message frequencies ranging from 15 ms to 2 s [1, 2, 29, 30]; we pick two points in this range: 15 ms as a lower bound and 750 ms per the evaluation baseline [29].

We observe that, even at high loads, DPTP-OW can achieve 10 ns bounds in both precision (Figure 12a) and jitter (Figure 12b) without imposing on user traffic. These bounds are similar to or better than DPTP, which incurs high-priority bandwidth overhead. Preliminary tests on higher-link speeds indicate that precision will only improve as τ decreases. In Figure 12b, we further observe that selective synchronization is an effective technique to reduce the message complexity of the protocol while maintaining low jitter and good precision.

4.3 Use Case #3: Congestion Feedback

Finally, many modern networks rely on robust load balancing algorithms to efficiently utilize their multiple paths. There are numerous approaches to load balancing, but among them,

adaptive approaches [3, 26] are attractive as they can react to current network conditions when making balancing decisions.

A state-of-the-art approach is taken by HULA [26], which proposes a protocol for adaptive data center load balancing using programmable switches. In HULA, every switch maintains two tables: a `bestHop` table that stores the best next-hop to each destination ToR, and a `pathUtil` table that stores the utilizations of those next-hops. Destination ToRs periodically flood the network with high-priority probes that traverse all paths (in the reverse direction, dst-to-src) and track the bottleneck link utilization of the best such path—intermediate switches update their `bestHop`/`pathUtil` tables accordingly.

As in the previous use cases, congestion feedback mechanisms like the one in HULA force a tradeoff between overhead and the availability/freshness of congestion data. HULA eventually sets the probing interval to 1-RTT and makes a case for why that is a good tradeoff, but OrbWeaver can potentially provide similar performance using only opportunistic sends.

4.3.1 An OrbWeaver Redesign

An OrbWeaver-inspired redesign replaces the high-priority HULA probes with OrbWeaver's opportunistic IDLE packets. There are two new challenges. The first is building a 'flood' communication model on top of OrbWeaver's opportunistic sends. The second is dealing with congestion on the reverse path and the resulting lack of new information.

Per-path propagation: For any path through the network, there are two types of hops: ingress-to-egress hops (that bridge the pipelines of a local switch) and egress-to-ingress hops (that bridge adjacent switches).

For the former, HULA-OW leverages the switching ASIC's PCIe interface to asynchronously mirror the `pathUtil` table between the ingress and egress pipelines of a single switch. We use Mantis [47] to mirror the registers, which completes a mirror operation every ~20 µs without impacting data plane throughput. For the latter type of hop, the system simply sends the contents of `pathUtil` using IDLE packets. To make this process more efficient, we can stripe the `pathUtil` table across m registers and pack m (`dstToR`, `pathUtil`) records into each IDLE packet round-robin style. In an unloaded network, the full table is transmitted in $\frac{R\tau}{m}$ time, where R is the number of ToRs in the data center. We note that even for $R = 1000$ and $m = 1$ (i.e., an unoptimized update rate), this is still more frequent than HULA.

Stale information: If there is persistent congestion on the reverse path, utilization information may not be able to propagate across the network; the switch adjacent to the congestion will know the utilization of the adjacent link, but not downstream links. To handle this case, HULA-OW uses a simple aging mechanism. Specifically, it will track the EWMA of all observed `pathUtil` values for every destination ToR (in addition to the minimum). After each RTT with no information from the best path, it will shift the best path's `pathUtil` value toward the average (with a lower bound of the adjacent link's
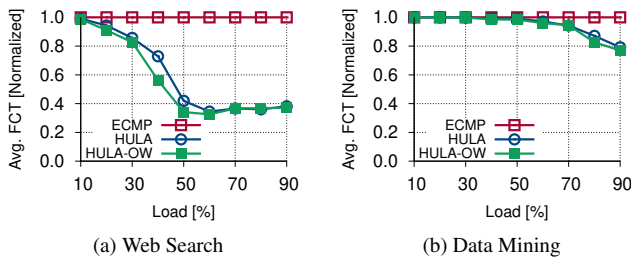
Figure 13: Avg. FCT (normalized to ECMP) for HULA and HULA-OW upon different loads of DCTCP and VL2 traces.

utilization). If no information comes from *any* neighbor for several RTT and the adjacent links are all equal, the switch will fall back to random flowlet placement.

Dealing with a lack of opportunity to send. We note that the effect of the above metric-aging strategy is that `bestHop` will be quickly overwritten by the 'next-best hop' whose reverse path has opportunities to send. Assuming that at least some congestion information gets through, HULA-OW will still provide substantial benefits due to properties like the power of two choices [34]. In the worst case, it achieves equivalent performance to flowlet ECMP.

Benefits. Across all regimes, HULA-OW eliminates the probe overhead on network bandwidth. In networks with low utilization or high burstiness, it provides more frequent utilization updates than HULA in addition to increasing the peak usable bandwidth (see below).

4.3.2 Evaluation

Performance. We evaluate HULA-OW in NS-2 using the same FatTree topology as the original paper (Figure 4 of [26]). Also like HULA, we leverage synthetic workloads based on web-search [4] and data-mining [16]) and configure HULA to probe at a 200 µs interval. Figure 13 shows the avg. FCT (normalized to ECMP) for HULA and HULA-OW.

Despite the frequent periods of full utilization in these workloads (especially at high average load), we observe that HULA-OW is able to find sufficient gaps between packets to efficiently transfer utilization information. Overall, HULA-OW is able to provide comparable or better performance than HULA in all of the tested cases, even in the presence of very high average utilization. The performance is also always either equivalent or better than the ECMP baseline.

Overhead reduction. The bandwidth overhead of HULA probes is given by $\frac{probeSize \times numToRs \times 100}{probeFreq \times linkBandwidth}$ [26]. With 500 ToRs, probeFreq=200 µs, probeSize=64 B, and 100 Gbps links, HULA occupies 1.6% of the network's bandwidth. In contrast, HULA-OW occupies close to *zero* of the network's usable bandwidth and only 1.5% of the packet-level capacity of the ingress pipeline (which HULA's probes also consume).

5 Related Work

Leveraging unused resources. OrbWeaver is not the first system to propose the opportunistic use of leftover resources. Indeed, many applications of priorities are in a similar spirit. Even in contexts outside of computer networking, others have used low-priority background tasks and spot VMs to harvest unused CPU cycles and memory [5].

In networking, close related work includes software WANs like SWAN [21] and B4 [24], which divide traffic into classes that range from interactive to background—interactive traffic is given priority while background traffic soaks up any remaining bandwidth. These systems successfully provide opportunistic bandwidth utilization but focus on end-host data. As explained in Section 2, these approaches can leave parts of the network unutilized due to both application traffic patterns and structural bottlenecks. OrbWeaver is, thus, complementary to these approaches and can be used to reclaim the remaining bandwidth for intra-network coordination.

Prior work has also applied similar techniques to lower layers, for instance, in the case of Ethernet's IDLE symbols or F10's rapid heartbeats [32]. F10, in particular, proposed a failure detection mechanism that is close to OrbWeaver's in which devices continue to send traffic even when idle. In comparison, OrbWeaver's contribution is make the idea practical on high-speed programmable switches, to closely examine the resulting impacts on switch configurations and user traffic, and to show how to seamlessly integrate the weaved stream into a spectrum of applications beyond the use case of F10.

Applications of OrbWeaver. OrbWeaver also builds explicitly on prior work that improves networks with coordination, signaling, and probes. We refer readers to the relevant parts of Section 4 for a discussion of the systems on which Orb-Weaver builds, and to the original papers for a more complete examination of related work for our applications.

In general, however, OrbWeaver improves on much of the prior work by providing comparable or better performance with near-zero overhead. Exceptions include systems like F10 [32] and DTP [29], which use hardware support to eliminate protocol overheads. As mentioned above, OrbWeaver's contribution is to generalize the concept and demonstrate a practical framework for it on commodity network devices.

6 Conclusion

Must data plane applications always choose between coordination fidelity and bandwidth overhead? This paper demonstrates that, somewhat surprisingly, they do not. To that end, we introduce OrbWeaver, a framework for opportunistic coordination in a manner that does not affect user traffic or switch power consumption. Using three recently proposed systems, we show how to leverage OrbWeaver to eliminate their bandwidth overheads while maintaining their efficacy.

Acknowledgments

References

[1] Ieee standard 1588-2008. https://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4579757, 2008.

[2] Juniper precision time protocol overview. https://www.juniper.net/documentation/us/en/software/junos/time-mgmt/topics/concept/ptp-overview.html, 2020.

[3] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, et al. Conga: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 503–514, 2014.

[4] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 63–74, 2010.

[5] Pradeep Ambati, Inigo Goiri, Felipe Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, Marcus Fontoura, and Ricardo Bianchini. Providing slos for resource-harvesting vms in cloud platforms. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 735–751. USENIX Association, November 2020.

[6] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang (Harry) Liu, Jitu Padhye, Boon Thau Loo, and Geoff Outhred. 007: Democratically finding the cause of packet drops. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 419–435, Renton, WA, April 2018. USENIX Association.

[7] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. Pint: Probabilistic in-band network telemetry. SIGCOMM '20, page 662–680, New York, NY, USA, 2020. Association for Computing Machinery.

[8] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280, 2010.

[9] Caida. The caida ucsd statistical information for the caida anonymized internet traces. https://www.caida.org/data/passive/passive_trace_statistics.xml, 2019.

[10] M. Chiesa, R. Sedar, G. Antichi, M. Borokhovich, A. Kamisiński, G. Nikolaidis, and S. Schmid. Fast reroute on programmable switches. *IEEE/ACM Transactions on Networking*, pages 1–14, 2021.

[11] Marco Chiesa, Roshan Sedar, Gianni Antichi, Michael Borokhovich, Andrzej Kamisiński, Georgios Nikolaidis, and Stefan Schmid. Purr: A primitive for reconfigurable fast reroute: Hope for the best and program for the worst. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, pages 1–14, 2019.

[12] Intel Corporation. P4-16 intel tofino native architecture – public version. Application Note 631348-0001, Intel Corporation, March 2021.

[13] Thomas G. Edwards and Warren Belkin. Using sdn to facilitate precisely timed actions on real-time data streams. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, page 55–60, New York, NY, USA, 2014. Association for Computing Machinery.

[14] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 81–94, 2018.

[15] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM 2011 conference*, pages 350–361, 2011.

[16] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, pages 51–62, 2009.

[17] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 139–152, 2015.

[18] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 357–371, New York, NY, USA, 2018. Association for Computing Machinery.

[19] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, pages 71–85, 2014.

[20] Thomas Holterbach, Edgar Costa Molero, Maria Apostolaki, Alberto Dainotti, Stefano Vissicchio, and Laurent Vanbever. Blink: Fast connectivity recovery entirely in the data plane. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 161–176, 2019.

[21] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 15–26, New York, NY, USA, 2013. Association for Computing Machinery.

[22] Kuo-Feng Hsu, Ryan Beckett, Ang Chen, Jennifer Rexford, and David Walker. Contra: A programmable system for performance-aware routing. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 701–721, 2020.

[23] Van Jacobson. Compressing tcp/ip headers for low-speed serial links. Technical report, RFC 1144, February, 1990.

[24] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 3–14, New York, NY, USA, 2013. Association for Computing Machinery.

[25] Pravein Govindan Kannan, Raj Joshi, and Mun Choon Chan. Precise time-synchronization in the data-plane using programmable switching asics. In *Proceedings of the 2019 ACM Symposium on SDN Research*, pages 8–20, 2019.

[26] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. Hula: Scalable load balancing using programmable data planes. In *Proceedings of the Symposium on SDN Research*, pages 1–12, 2016.

[27] Changhoon Kim, Anirudh Sivaraman, Naga Katta, Antonin Bas, Advait Dixit, and Lawrence J Wobker. In-band network telemetry via programmable dataplanes. In *Demo paper at SIGCOMM '15*, 2015.

[28] Daehyeok Kim, Jacob Nelson, Dan R. K. Ports, Vyas Sekar, and Srinivasan Seshan. Redplane: Enabling fault-tolerant stateful in-switch applications. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 223–244, New York, NY, USA, 2021. Association for Computing Machinery.

[29] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. Globally synchronized time via datacenter networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 454–467, 2016.

[30] Yuliang Li, Gautam Kumar, Hema Hariharan, Hassan Wassel, Peter H Hochschild, Dave Platt, Simon Sabato, Minlan Yu, Nandita Dukkipati, Prashant Chandra, et al. Sundial: Fault-tolerant clock synchronization for datacenters. 2020.

[31] Junda Liu, Aurojit Panda, Ankit Singla, Brighten Godfrey, Michael Schapira, and Scott Shenker. Ensuring connectivity via data plane mechanisms. In *10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pages 113–126, 2013.

[32] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. F10: A fault-tolerant engineered network. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 399–412, 2013.

[33] Jim Martin, Jack Burbank, William Kasch, and Professor David L. Mills. Network Time Protocol Version 4: Protocol and Algorithms Specification. RFC 5905, June 2010.

[34] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[35] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 85–98, 2017.

[36] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A centralized "zero-queue" datacenter network. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 307–318, New York, NY, USA, 2014. Association for Computing Machinery.

[37] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. Inside the social network's (datacenter) network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 123–137, 2015.

[38] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. *SIGCOMM Comput. Commun. Rev.*, 45(4):183–197, August 2015.

[39] John Sonchack. *Balancing Performance and Flexibility in Hybrid Network Telemetry Systems*. PhD thesis, University of Pennsylvania, 2020.

[40] John Sonchack, Devon Loehr, Jennifer Rexford, and David Walker. Lucid: A language for control in the data plane. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 731–747, New York, NY, USA, 2021. Association for Computing Machinery.

[41] Charles E. Spurgeon. *Ethernet: The Definitive Guide*. O'Reilly & Associates, Inc., USA, 2000.

[42] Nik Sultana, John Sonchack, Hans Giesen, Isaac Pedisich, Zhaoyang Han, Nishanth Shyamkumar, Shivani Burad, André DeHon, and Boon Thau Loo. Flightplan: Dataplane disaggregation and placement for p4 programs. In *18th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 21)*, pages 571–592, 2021.

[43] Maarten Van Steen and Andrew S Tanenbaum. *Distributed systems*. Maarten van Steen Leiden, The Netherlands, 2017.

[44] Xin Wu, Daniel Turner, Chao-Chih Chen, David A Maltz, Xiaowei Yang, Lihua Yuan, and Ming Zhang. Netpilot: automating datacenter network failure mitigation. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 419–430, 2012.

[45] Nofel Yaseen, John Sonchack, and Vincent Liu. Synchronized network snapshots. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 402–416, 2018.

[46] Nofel Yaseen, John Sonchack, and Vincent Liu. tpprof: A network traffic pattern profiler. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 1015–1030, Santa Clara, CA, February 2020. USENIX Association.

[47] Liangcheng Yu, John Sonchack, and Vincent Liu. Mantis: Reactive programmable switches. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 296–309, 2020.

[48] Lior Zeno, Dan RK Ports, Jacob Nelson, and Mark Silberstein. Swishmem: Distributed shared state abstractions for programmable switches. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, pages 160–167, 2020.

[49] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. High-resolution measurement of data center microbursts. In *Proceedings of the 2017 Internet Measurement Conference*, pages 78–85, 2017.

[50] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, et al. Flow event telemetry on programmable data plane. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 76–89, 2020.

[51] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, Lihua Yuan, Ming Zhang, Ben Y Zhao, et al. Packet-level telemetry in large datacenter networks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 479–491, 2015.

[52] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Klaus-Tycho Förster, Arvind Krishnamurthy, and Thomas Anderson. Understanding and mitigating packet corruption in data center networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 362–375, 2017.

Application Class	System	Weaved Inference?	IDLE Messaging?	Description
Traffic Engineering	Flowlet load balancing [3, 26]	✓	✓	Section 4.3.
	Performance-aware routing [22]		✓	Propagate route updates in customizable distance-vector routing algorithms using IDLE packets.
	Micro-burst detection [49]	✓	✓	Detect micro-bursts from weaved stream, provide feedback to upstream switches with IDLE packets.
Fault Tolerance	Fast failure recovery [50]	✓	✓	Detect failures (Section 4.1), alert upstream switches with IDLE packets for fast data-plane mitigation [10].
	Consistent replicas [28, 48]		✓	Synchronize eventually-consistent distributed state, e.g., for distributed firewalls, with IDLE packets.
Monitoring	Packet forensics [19]		✓	Transfer packet postcards in IDLE packets to reduce overhead of packet history tracking.
	Network queries [18, 35]	✓	✓	Support queries over both flow and weaved stream statistics, export query results in IDLE packets.
	Latency localization [17]	✓	✓	Measure latency in network core using weaved stream, disseminate measurements with IDLE packets.
Network Services	Clock synchronization [25]	✓	✓	Section 4.2.
	Header compression [23, 42]		✓	Synchronize state of point-to-point packet header compressors with IDLE packets.
	Event-based network control [40]		✓	Carry network control events in IDLE packets.

Table 2: OrbWeaver use cases. A diverse range of data-plane applications can use OrbWeaver's weaved stream to learn about conditions in the network and/or communicate via IDLE packets that consume no data-packet bandwidth.

A Applications of OrbWeaver

Table 2 surveys 11 applications that can benefit from an Orb-Weaver implementation, belonging to four distinct classes. We describe several implementations in Section 4. All applications can be expressed as OrbWeaver P4 programs with the basic architecture shown in Figure 8.

Across all applications, we find that there are two overarching benefits to an OrbWeaver implementation:

1. OrbWeaver's weaved stream allows data plane applications to infer information about network conditions, such as the presence of congestion or failures in an upstream path.

2. OrbWeaver's IDLE packet abstraction lets data plane applications disseminate information without consuming user bandwidth. IDLE packets are useful for data transfer between directly connected switches (e.g., to synchronize the context tables of a switch-to-switch packet-header compressor [42]) or across the wider network (e.g., to disseminate information about network faults [32], congestion [49], or even user query metrics [35]).

We note that our focus of these applications and this paper is in-network communication. However, end hosts may also be able to benefit from OrbWeaver, e.g., by examining the output of the weaved stream coming from host-facing ports of ToR switches. Efficient end-host generation of a weaved stream may also be possible, but we leave a full exploration to future work.

A.1 Balancing Multiple Applications

IDLE packets are generated and weaved entirely by the Orb-Weaver framework. Applications only embed information and extract it in the receiver. IDLE packets can carry the information of multiple applications. For example, a time synchronization application that needs 12B to carry 4 timestamps can co-exist with a failure detection protocol that needs 48B. In this paper, we assume minimum-sized packets but, in principle, IDLE packets can be MTU-sized with the only effect being a proportionally increased worst-case packet delay. Of course, there are fundamentally a limited number of bytes in each IDLE packet; OrbWeaver leaves the decision on how to allocate these bytes to network architects and operators.

A.2 Preventing Starvation

The primary goal of the paper is to explore the opportunistic use of IDLE cycles for in-network coordination. Because of our opportunistic approach, there may be cases where IDLE packets get starved by user packets; however, as previously noted, two factors mitigate the issue:

• The lack of IDLE packets itself reveals concrete information of the network condition (per R1 guarantee of the weaved stream predictability).

• Prior works observed that persistent user traffic is rare, instead, IDLE cycles (every 10s or 100s of μs) are ubiquitous.

A wide range of applications can be implemented with only opportunistic communication. Of course, some applications may need additional guarantees, e.g., applications requiring a strict, real-time guarantee w.r.t. minimum rate (i.e., maximum inter-IDLE-packet gap); or applications that need more aggregate bandwidth than the weaved stream can guarantee in a timely fashion.

In these cases, networks can apply a priority escalation mechanism by adding a single register of N (number of ports) slots and check the elapsed time since last seen IDLE packet. Applications can seamlessly escalate the priority of IDLE packets when too much time passes (per the applications' guaranteed rate SLO). In these situations, OrbWeaver still eliminates nearly all overhead in the presence of (micro)bursts, but may impose a fixed overhead during extended periods of congestion.

B   Generalization to Other Platforms

Our focus in this paper was on the Tofino family of programmable switches. While a detailed taxonomy and analysis of every programmable platform is out of the scope of this paper, there is reason to believe that other programmable platforms have similar features or can emulate the features needed to implement OrbWeaver.

In particular, OrbWeaver leverages three hardware features of Tofino switches: (1) packet generation, (2) multicast, and (3) packet prioritization. Among these, support for the latter two can be found in almost every modern forwarding device that is designed to handle the Ethernet protocol. Support for onboard packet generation is not as universal; however, one potential solution is to connect a port on each switch to a simple device/CPU responsible for generating regular, periodic packets. Of course, a CPU, even with real-time scheduling optimizations, may not be as dependable as the Tofino packet generator. This may necessitate additional tolerances.

Finally, our conversations with switch vendors indicate that OrbWeaver's mechanisms will scale to future switches with both increased bandwidth and port counts. Part of this is due to the fact that most of OrbWeaver's components scale with the clock rate of the switch and/or are independent to each pipeline. The notable exception is packet generation; however, we note that OrbWeaver currently has more than an order of magnitude of headroom (Section 3.2.1). If MTU transmission time does eventually outpace packet generation latency, OrbWeaver's properties will degrade gracefully.

C   Energy-Efficient Ethernet (EEE)

The Ethernet standard contains an optional EEE mechanism [41], which allows switches to transition links into a Low-Power Idle (LPI) mode when there is no data to send.

OrbWeaver may be able provide compatibility by turning off the IDLE stream on a per-port, per-direction basis if there is no user traffic during the past S seconds. Each packet flowing between two OrbWeaver switches would then need a single bit reserved as an 'LPI' indicator. Upon receiving an IDLE packet with the 'LPI' indicator set, a receiver will change its expectation from requiring a packet every τ_i seconds to requiring one every τ_i' seconds ($\tau_i' \gg \tau_i$). The very first user packet after the low-power idle mode will be sent with the 'LPI' indicator unset. Loss can be addressed by again emulating EEE and sending several indicator packets in a row.

Enabling this feature may impact the responsiveness of OrbWeaver applications, but we note that all of the use cases studied can make do with less frequent but still regular coordination. OrbWeaver may be able to synchronize these low-power updates with existing synchronization-maintenance events in the PHY.

D   Proof of Priority-effect on User Traffic

Theorem. For an arbitrary user packet size distribution and arrival process, with strict priority scheduling and a measurement time window $T \gg \Delta t$ (Δt denotes transmission time of a single IDLE packet), the throughput of the user traffic is unaffected by the IDLE stream.

Proof. Consider a packet sequence $p_1, \ldots, p_n$ with size $\Delta t_1, \ldots, \Delta t_n$ and original schedule $t_1, \ldots, t_n$, denote the new schedule upon the coexistence of IDLE stream as $t_1', \ldots, t_n'$.

We first prove $\forall i \in [1, n-1], t_i' \le (t_i + \Delta t) \to t_{i+1}' \le (t_{i+1} + \Delta t)$. The case for preemptive scheduling is trivially true. We focus on the case of non-preemptive scheduling.

Base case with p_1: the worst case delay of the transmission is when right at t_1, an IDLE packet is scheduled to transmit and with strict priority p_1 is scheduled right next to it. Hence $t_1' \le (t_1 + \Delta t)$.

For the inductive step, given the new schedule of p_i satisfying $t_i' \le (t_i + \Delta t)$, we need to show that $t_{i+1}' \le (t_{i+1} + \Delta t)$. There are three cases for the next packet p_{i+1}:

- $t_{i+1} > (t_i + \Delta t_i + \Delta t)$: at t_{i+1}, the previous packet has finished transmission in the new schedule since $t_{i+1} > (t_i + \Delta t_i + \Delta t) \ge t_i' + \Delta t_i$. The worst case delay is when IDLE packet is scheduled right at t_{i+1} and the transmission is delayed by Δt, i.e., $t_{i+1}' \le (t_{i+1} + \Delta t)$ holds.

- $t_i' + \Delta t_i \le t_{i+1} \le (t_i + \Delta t_i + \Delta t)$: at t_{i+1}, p_i finishes transmitting in the new schedule, similar to the previous case, the worst case is Δt when right at t_{i+1}, IDLE packet gets scheduled, hence $t_{i+1}' \le (t_{i+1} + \Delta t)$ holds.

- $t_i + \Delta t_i \le t_{i+1} < t_i' + \Delta t_i$: p_{i+1} has been queued since p_i is still transmitting until $t_i' + \Delta t_i$ in the new schedule. With strict priority, p_{i+1} will start transmission right at $t_i' + \Delta t_i$ ignoring the IDLE packet. Hence, $t_{i+1}' = t_i' + \Delta t_i \le t_i + \Delta t + \Delta t_i \le (t_{i+1} + \Delta t)$.

Configuration	SRAM	TCAM	Metadata	Tbls	Regs
16×100 Gbps	80 KB	1.28 KB	85 b	3	1
32×25 Gbps	80 KB	1.28 KB	53 b	3	1

Table 3: Additional data plane resources for OrbWeaver's weaved stream generation over an L2 forwarding switch. Ports are binned into groups of 2 and 4, and only 256 multicast groups reserved.

By induction, we have $t'_n \leq (t_n + \Delta t)$, that is, the latency impact is tightly bounded by Δt for an arbitrary user packet and won't accumulate across packets. Given such fixed workload, consider the impact of the IDLE stream over the original transmission time $T = t_n + \Delta t_n - t_1$. For the new transmission time window $[t'_1, t'_n + \Delta t_n]$, the duration $T' = t'_n + \Delta t_n - t'_1 \leq \max(t'_n) + \Delta t_n - \min(t'_1) \leq t_n + \Delta t + \Delta t_n - t_1$. Hence, $T' - T \leq \Delta t$. Since $T \gg \Delta t$, the throughput of the high priority user packet stream is not impacted. □

E   Probability of Notification in Use Case #1

We can formally express the probability that a notification is sent before the flow is evicted. Consider the case where there is a drop in flow f and user packets are all MTU-sized, i.e., there is one packet per period, τ. Assume that the flow cache holds N records and 3 can be packed in each IDLE.

$$P(\text{notified}) = \frac{P(\text{IDLE contains } f)}{P(\text{IDLE contains } f) + P(\text{new } f' \text{ replaces } f)}$$

$$= \frac{\frac{3}{N}P(\text{IDLE})}{\frac{3}{N}P(\text{IDLE}) + \frac{1}{N}(1 - P(\text{IDLE}))P(\text{new flow})}$$

$$= \frac{P(\text{IDLE})}{P(\text{IDLE}) + (1 - P(\text{IDLE}))P(\text{new flow})/3}$$

where $P(\text{IDLE})$ is the probability that an IDLE packet was sent during a given period τ, and $P(\text{new flow})$ is the probability that a user packet's flow cannot be found in the cache. Smaller packets multiply the second term in the denominator; a larger N decreases it by improving cache hit rates. The probability that a flow record is evicted *before* it is sent (i.e., that we miss the loss) is 1 less the above value.

F   OrbWeaver Data Plane Resource Overhead

Section 3 details the overhead of OrbWeaver's weaved stream generation on user traffic and energy usage. We note that OrbWeaver also uses data plane resources for IDLE seed packet filtering and replication, as shown in Table 3. For each category, OrbWeaver only occupies a small fraction of the total switch resources (for instance < 1% of both SRAM and TCAM).

CloudCluster: Unearthing the Functional Structure of a Cloud Service

Weiwu Pang
University of Southern California

Sourav Panda
University of California, Riverside

Jehangir Amjad
Google Inc.

Christophe Diot
Google Inc.

Ramesh Govindan
University of Southern California

Abstract

In their quest to provide customers with good tools to manage cloud services, cloud providers are hampered by having very little visibility into cloud service functionality; a provider often only knows where VMs of a service are placed, how the virtual networks are configured, how VMs are provisioned, and how VMs communicate with each other. In this paper, we show that, using the VM-to-VM traffic matrix, we can unearth the *functional structure* of a cloud service and use it to aid cloud service management. Leveraging the observation that cloud services use well-known design patterns for scaling (*e.g.,* replication, communication locality), we show that *clustering* the VM-to-VM traffic matrix yields the functional structure of the cloud service. Our clustering algorithm, CloudCluster, must overcome challenges imposed by scale (cloud services contain tens of thousands of VMs) and must be robust to orders-of-magnitude variability in traffic volume and measurement noise. To do this, CloudCluster uses a novel combination of feature scaling, dimensionality reduction, and hierarchical clustering to achieve clustering with over 92% homogeneity and completeness. We show that CloudCluster can be used to explore opportunities to reduce cost for customers, identify anomalous traffic and potential misconfigurations.

1 Introduction

As more online services migrate to the cloud, and as the user base of these services increases, the complexity and scale of cloud deployments has increased significantly. Today, cloud services routinely use tens of thousands of VMs, geographically dispersed for reliability and low-latency access to customers. Monitoring and managing a cloud deployment can be significantly challenging, since the performance, cost, and reliability of the service can depend on a large number of factors: how the cloud customer maps logical functionality to VMs, how the VMs are provisioned, where they are located, how well the paths between the VMs are provisioned, and so on. More generally, how well a cloud service works depends both on how well a customer designs the service, and how well the provider provisions the underlying infrastructure.

Cloud service monitoring. Cloud providers struggle to provide customers with insights on the performance and reliability of a cloud service. This is because, while a VM provides a very convenient abstraction for computing and communication, the provider has (by design) very little *visibility* into cloud service logic embedded in the VM. This lack of visibility prevents providers from being able to relate problems observed at the service level to issues in the underlying infrastructure. For a given service, a provider often only knows where the VMs are, how much compute and storage each VM is provisioned with, customer-supplied names for the VMs, and how much traffic each VM exchanges with other VMs in the service. Customers are often loath to reveal more, for business and privacy reasons.[1]

Today, major cloud providers (such as Amazon Web Services [12], Azure Cloud [2] and Google Cloud Platform [3]) provide customers with monitoring services. Their monitoring services (AWS CloudWatch [12], Azure Monitor [2] and Google Cloud Monitoring [3]) expose, using customizable dashboards, metrics capturing the state and activity of the cloud service's VM instances (*e.g.,* their CPU and disk utilization, and the volume of network traffic to and from instances) as visible to the cloud provider, as well as other measures of the underlying networking infrastructure (*e.g.,* loss rates between instances). Some of these monitoring services also provide custom alerting mechanisms. Customers can define metrics that capture user-perceived performance, and configure alerts when these metrics exceed service-level objectives that cloud customers have with *their* customers.

Goal. Given that cloud service monitoring provides a competitive advantage, cloud providers continuously seek to add

---

[1]**Ethical considerations:** For the 15 cloud projects we used in the evaluations in the paper (§4), we obtained explicit consent. For each project, we only used information available to the cloud provider: VM locations, VM names, and the VM-to-VM traffic matrix. We used the VM-to-VM traffic matrix to generate the clusters, and names and locations to evaluate the performance of CloudCluster. After our evaluations, we shared the results with each customer, and obtained feedback.

innovative capabilities to their monitoring systems, despite their limited visibility into cloud services. In this paper, we describe a new capability not, to our knowledge, previously considered in the literature: inferring the *functional structure* of a cloud service, *i.e.,* how a cloud service is modularized across its many VMs. Our work is inspired by a body of prior work on inferring structural relationships between components in a distributed system (§6).

To explain what we mean by functional structure, consider Figure 1(a) which shows the connectivity graph of VMs (which VMs communicate with which other VMs) of a cloud service. These VMs reside in different cloud regions (roughly, parts of a continent, see §2); most communication is within a region, but some communication exists across regions. With just the information that the cloud provider has, it can only obtain this kind of a view of the project. Now suppose that this cloud service is, in fact, architected as in Figure 1(b): it has a front-end load-balancer and a backend processing layer. With the information she has, the cloud service operator's conceptual view of the structure of the service might be as shown in Figure 1(c): its VM instances are spread across two regions, with load balancer VMs (in red on the cluster on the left, and green on the cluster on the right) communicating with the processing-layer VMs (in magenta and cyan respectively) but not with other load balancer VMs, and the processing-layer VMs in each region communicating with each other as well. In addition, one of the load-balancers communicates with processing VMs in the other region (*e.g.,* due to overload in its own region).

The focus of our paper is to *unearth* the structure in Figure 1(c) *only using information available to the provider*. Specifically, we aim to develop methods that can extract this structure in which VMs are grouped into *VM groupings* by function and location. Ultimately, this will enable the provider to represent the service by a compact inter-grouping graph abstraction (Figure 1(d)).

In deriving the representations shown in Figure 1(c-d), CloudCluster can only determine that VMs in a cluster likely perform the same function, but cannot tell *which* function they perform (for example, whether the VMs run load-balancers, or image transcoders). This mitigates any privacy concerns cloud customers might have. Even so, we expect that in an actual deployment, a cloud provider will obtain consent from the customer before applying CloudCluster to the customer's service.

Approach. We hypothesize that we should be able to infer VM groupings by *clustering the VM-to-VM traffic matrix of a cloud service*. Clustering a traffic matrix implies grouping together similar rows; two rows are similar if the traffic from their corresponding VMs to all other VMs is similar. Intuitively, two functionally similar VMs are likely to satisfy this property. For example, how two load-balancer VMs in a region are likely to communicate with all other processing layer VMs in the same region is likely to be similar, so clustering
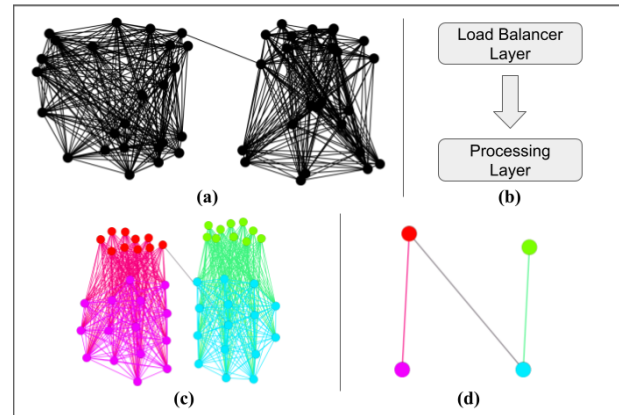
Figure 1: An example of different views of a cloud service: (a) VM connectivity graph as visible to the cloud provider; (b) The service architecture; (c) VM connectivity graph colored by function and location (the desired output of CloudCluster); (d) A compact inter-grouping graph abstraction.

will group them together. Furthermore, we expect clusters to be large because of the horizontal scaling employed by cloud services, which replicate processing or storage at a given layer using functionally identical VMs (*e.g.,* databases, in-memory stores, image transcoders *etc.*).

Challenge. Analyzing large VM-to-VM traffic matrices of real-world cloud services presents two challenges: scale, and robustness to variability and noise. At the scale of tens of thousand of VMs, any analysis must overcome the curse of dimensionality [60]; the sparsity of the traffic matrices in these higher-dimensions makes it difficult to derive insights from the data. Moreover, cloud services often vary in VM-to-VM traffic by several orders of magnitude, and methods of inferring their properties must accommodate this variability and be robust to noise introduced by the underlying measurement methodology (*e.g.,* by traffic sampling).

Contributions. This paper shows that *clustering* the VM-to-VM traffic matrix of a cloud service provider can help determine the functional organization of a cloud service, and that these clusters can be a useful abstraction for providing cloud customers with actionable insights into their service. To this end, the paper makes three contributions.

First, we develop a clustering algorithm, CloudCluster, that clusters VMs by similarity in their network communication characteristics (§3.4). CloudCluster is a novel combination of techniques, some known, and others new, to address the scaling and robustness challenges mentioned above. At its core, it uses a variant of *hierarchical clustering*, called agglomerative clustering [48] to determine clusters. This approach clusters VMs by proximity in some high-dimensional space. It requires a way to determine distance thresholds, and CloudCluster determines these thresholds dynamically in a data-driven manner. To scale better, it employs dimensionality reduction, and to be robust to variability in traffic volumes it scales traffic features (see §3 for more details).

Second, by evaluating the resulting clusters on 15 different

cloud service *projects*[2] (§4), we experimentally demonstrate that the resulting clusters *group together VMs by location and function*: *i.e.,* all VMs in a cluster are geographically co-located, and they perform the same function[3]. We verify this on cloud services that name VMs by function; for these projects, CloudCluster has homogeneity and completeness scores (metrics equivalent to precision and recall, respectively) of over 0.92 and 0.94 respectively.

Third, we demonstrate ways in which CloudCluster can be used to provide customers with actionable insights (§5). CloudCluster can analyze the inter-cluster graph (Figure 1(d)) to identify opportunities for reconfiguring VM placements to reduce cost: in one case, we found opportunities to reduce cost by 41.2% by provisioning an additional cluster to minimize inter-region traffic. It can also be used to detect anomalous traffic between clusters, to identify traffic shifts within a cloud project, or structural changes in the project across time. From 25 traffic anomalies reported either by an internal anomaly detector or the customer, a CloudCluster-based anomaly detector detected every anomaly, and identified the impacted clusters. CloudCluster can be used to detect potentially mis-labeled VM names (names that do not reflect function) or mis-provisioned VMs. In some projects, up to 1% of VMs appear to be mis-provisioned. In others, over 7% a project's VMs appear to be mis-labeled — their traffic patterns differ from the majority of VMs that have the same labels.

2 Anatomy of a Cloud Service

In this section, we provide a brief background on the structure of cloud services. Our description focuses on Google's cloud services; different service offerings may differ from this description in the details.

Google's cloud resources are hosted in multiple locations worldwide. The network is subdivided into *regions* which are in turn divided into *zones* [9]. A region represents a part of a continent, and zones represent disjoint geographical areas within a region in which infrastructure resides. This partitioning permits cloud customers to coarsely control the placement of VMs to, for example, ensure low-latency access to customers, control cost and ensure high availability.

Customers can organize their cloud service into *projects* [4], which are granular functional groupings that simplify management of a cloud service. For example, an ad-supported social media service can have different projects for the user-facing front-end, the ads subsystem, and an analytics backend. Depending on the scale of the service, projects can be large, spanning tens of thousands of VMs across multiple regions. In this paper, we focus on the structure of projects.

VMs in a project are connected by one or more *virtual*

networks [6] that provide isolation. Customers can organize VMs into *sub-networks* [7]: VMs in one sub-network must all be within the same region, and communicate over the same virtual network. Sub-networks simplify VM management tasks: *e.g.,* IP address assignment.

Customers populate projects with VMs. To create a VM, the customer: (a) selects a *configuration* for the VM (configurations differ in compute and storage), (b) specifies the VM's *name* (the name is opaque to the provider, but customers may embed hierarchical structure into a name; some customers name VMs by function, a feature we leverage in evaluating CloudCluster in §4), (c) identifies the sub-network and the virtual network the VM uses, and (d) specifies the region and zone the VM is located in. This is the *only* information a cloud customer explicitly provides to Google. In addition, if customers opt in to flow logging [10], the logging service records VM-to-VM traffic for each enabled project.

3 CloudCluster Design

In this section, we describe CloudCluster, whose goal is to discover the underlying structure of a cloud project. We discuss how it scales to large cloud projects, while being robust to noise and variability.

3.1 Goals, Approach, and Overview

Notation. The input to CloudCluster is a VM-to-VM traffic matrix for a cloud project, containing traffic volumes between each VM over a fixed aggregation window.[4] Traffic volumes are obtained by sampling flows. In §4, we discuss the actual values of the aggregation window and the sampling frequency. Formally, we denote this traffic matrix by $\mathbf{Y}$, with dimensions $n \times m$, where n is the number of source VMs (belonging to this specific project under consideration) and m is the number of destination VMs (which do not all have to belong to the same project, since VMs in a project can communicate with external clients or VMs in other projects).[5] The i, j-th entry y_{ij} of $\mathbf{Y}$ represents the volume of traffic (in bytes) from VM i to VM j, where $i \in [n], j \in [m]$.

Challenge: Noise. Since $\mathbf{Y}$ is sampled, it is bound to be noisy. Aside from the error induced by sampling, measurement errors and randomness in traffic patterns can also induce noise. To model this, we can write:

$$\mathbf{Y} = \mathbf{M} + \mathbf{E} \tag{1}$$

---

[2]As discussed in §2, a project is a granular functional grouping within a cloud service.

[3]In this paper, we use the term *function* to denote a long-lived heavy-weight service that forms part of a cloud service; we do *not* consider services deployed using ephemeral cloud functions (*e.g.,* lambdas).

[4]CloudCluster uses minimum possible aggregated information, namely the communication volume. Other metadata (*e.g.,* port numbers, process names) might be helpful in finding the functional structure. CloudCluster does not use this information. With consent from the customer, it might be possible to use this to improve our clustering, but we have left it to future work, in part because it is not clear whether customers will consent to revealing additional information.

[5]CloudCluster does not currently model traffic to cloud native services, like traffic to Google Cloud Storage [5]). Identifying traffic volumes from these services requires using other instrumentation services (*e.g.,* storage service logs), and we have left this to future work.

---

where $\mathbf{M}$ is the unobserved noise-free, *true* traffic matrix and $\mathbf{E}$ is a noise matrix. We assume e_{ij} is independent of all other entries and $\mathbb{E}[e_{ij}] = 0$ and $Var[e_{ij}] < \infty, \forall i \in [n], \forall j \in [m]$. This implies that $\mathbf{M} = \mathbb{E}[\mathbf{Y}]$.

Challenge: Scale. $\mathbf{Y}$ can be large, since projects can have tens of thousands of VMs. We have observed, through manual inspection of cloud projects, that to enable projects to scale, designers often group VMs that perform similar functions. At the front-end, load-balancers redirect requests to VMs that scale with the request load; all these VMs perform the same function (*e.g.,* handle requests). In turn, at the back-end, these VMs may invoke other services that may be replicated across several identical VMs, or may send the request to a coordinator VM that invokes an iterative distributed computation spread across several identical VMs. Such structures result in *VM groupings*. We hypothesize that VMs in a group have similar traffic patterns (in terms of which VMs they communicate with, and the volume of traffic). If this hypothesis is true, then $\mathbf{M}$ must be a *low rank matrix*.

We can formalize a VM grouping as:

Definition 1 *VM Grouping. Let a VM Grouping be denoted by $\mathbf{S}_i$. Let m_u denote a row of matrix, $\mathbf{M}$. m_u belongs to $\mathbf{S}_i$, if*

$$d(m_u, m_v) < \min_r\{d(m_u, m_r)\}, \forall v \in \mathbf{S}_i, \forall r \notin \mathbf{S}_i$$

$$\wedge d(m_u, m_v) < \delta, \forall v \in \mathbf{S}_i$$

$d(\cdot, \cdot)$ *is some distance function and δ is a distance threshold.*

Definition 1 implies that *similar* rows will be grouped together if they are most similar to each other and their similarity, quantified via a distance function, $d(\cdot, \cdot)$, is less than the distance threshold. In practice, this threshold can be different for different cloud projects.

Goal and Approach. Our goal is to discover all VM Groupings present in a cloud project. To do this, CloudCluster (a) estimates $\mathbf{M}$ and then (b) clusters VMs (rows of $\mathbf{M}$ with similar traffic patterns) to find the VM groupings.

To estimate $\mathbf{M}$, we leverage prior work, such as [27] and [21], which show that in a setting like ours, we can estimate well and with consistency the low-rank and noise-free, but unobserved, matrix, $\mathbf{M}$, from a random observation of the noisy matrix $\mathbf{Y}$, where $\mathbf{M} = \mathbb{E}[\mathbf{Y}]$.

Clustering algorithm: Overview. Using the estimate $\hat{\mathbf{M}}$, CloudCluster's clustering algorithm[6] seeks to extract VM Groupings according to Definition 1, with $\mathbf{M}$ replaced by $\hat{\mathbf{M}}$. It must also address the scalability and robustness challenges identified above. To do this, CloudCluster's algorithm has four components (Algorithm 1): 1) Feature scaling to transform the input traffic matrix. 2) Matrix estimation to estimate $\mathbf{M}$. 3) Hierarchical clustering to group similar VMs. 4) Cluster merging to fuse similar clusters. We describe each component in the following subsections.

---

[6]This is orthogonal to prior work that has explored clustering to group similar traffic matrices [59].

Algorithm 1: Steps in VM clustering
input : $\mathbf{Y}$, threshold θ to merge similar clusters
output: Clusters *merged_clusters*
1 scaled_$\mathbf{Y}$ = feature_scaling($\mathbf{Y}$);
2 scaled_$\hat{\mathbf{M}}$ = TruncatedSVD(scaled_$\mathbf{Y}$);
3 clusters = hierarchical_clustering(scaled_$\hat{\mathbf{M}}$);
4 merged_clusters = merging(clusters, scaled_$\hat{\mathbf{M}}$, θ);

3.2 Feature Scaling

What is a feature and why scaling is necessary. Each row of $\mathbf{Y}$ can be treated as a (high-dimensional) feature. Then, identifying similarity in this feature space is equivalent to identifying VMs that have similar traffic patterns.

In practice, even within a single project, traffic volumes between VMs can span several orders of magnitude. This can make it difficult to discriminate between low and medium volume traffic patterns. Clustering relies on a distance metric, and many applicable distance metrics are disproportionately sensitive to larger values.

Log Scaling. Feature scaling normalizes the range of each feature to enable clustering algorithms to be robust to highly variable traffic volumes. Of the existing feature scaling methodologies, standardization and minmax scaling cannot handle the range of traffic volumes we see in cloud projects. Standardization replaces each feature's value by how many standard deviations it is above or below the mean [1]. Minmax scaling transforms each individual feature value into the ratio between that value's distance from the minimum to the range of values [1]. Traffic in cloud projects can span several orders of magnitude (from 0 to 10^9) and have skewed distributions; linear transformations like minmax scaling, or those that assume Gaussianity, like standardization, do not work well. For this reason, we choose *log scaling*, which uses the natural logarithm of the traffic instead of the original values; this handles volume variability much better (we demonstrate this experimentally in §4.4).

3.3 Estimating M

Estimating singular values. Singular value thresholding can produce a good estimate, $\hat{\mathbf{M}}$, of the low-rank $\mathbf{M}$, using only observations from $\mathbf{Y}$ (see [27], [21], [20]). However, estimating the number of singular values to keep cannot be determined exactly. After performing Singular Value Decomposition of the matrix, we choose the number of singular values to retain based on an *elbow* finding heuristic such as one introduced by [51]. The *elbow* suggests the approximate number of singular values to retain because most of the singular values after the *elbow* contribute little to the spectrum of the matrix. Figure 2 shows the spectrum of singular values for a traffic matrix for a project with over 3000 VMs. The sharp decline in the spectrum after about 50 singular values is indicative of a low-rank structure. Singular values in the tail which don't quite decay to 0 indicate random noise (which tends to spread
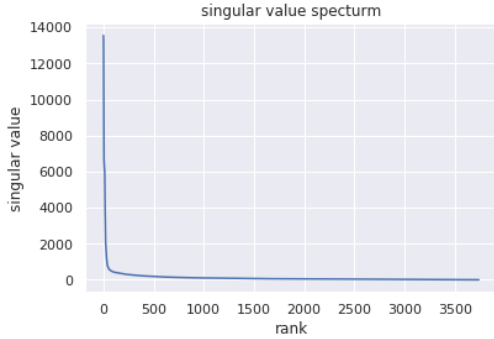
Figure 2: Singular Value spectrum of a traffic matrix with dimension (3742x3271)

across all orthogonal directions) with small finite variance (indicated by the small magnitude).

Extracting an r-rank approximation of M. Once the number of singular values r is heuristically determined, performing an SVD produces the reduced rank estimate of the original matrix. Specifically, given the $n \times m$ original traffic matrix $\mathbf{Y}$, SVD produces the reduced dimension matrix $\hat{\mathbf{M}}$, such that:

$$\hat{\mathbf{M}} = \mathbf{U_r}\mathbf{\Sigma}_r\mathbf{V_r^T},$$

where $\mathbf{\Sigma}_r$ is an $r \times r$ diagonal matrix of the singular values of $\mathbf{Y}$, $\mathbf{U_r}$ and $\mathbf{V_r}$ are orthonormal bases of dimensions $n \times r$ and $m \times r$, respectively. $\hat{\mathbf{M}}$ is a low-rank, *i.e.,* rank = $r \ll \min\{n,m\}$, approximation to the original matrix. However, $\hat{\mathbf{M}}$ is of dimensions $n \times m$. We need to project this matrix to an $n \times r$ subspace which will allow us to retain all the rows (associated individually to VMs), each of r-dimensional feature (columns). We denote this desired matrix by $\hat{\mathbf{M}_r}$, determined by:

$$\hat{\mathbf{M_r}} = \mathbf{U_r}\mathbf{\Sigma}_r$$

Effectively, the retained r singular values of the original matrix $\mathbf{Y}$ determine how to scale each of the $r-$dimensional orthonormal vectors in $\mathbf{U_r}$. $\hat{\mathbf{M_r}}$ remains a good approximation of $\mathbf{M}$ (in a reduced dimensional subspace) because it is simply a projection of each of the rows in $\hat{\mathbf{M}}$ (which is the best rank-r approximation of $\mathbf{M}$) on to a r-dimensional subspace. Both $\hat{\mathbf{M}}$ and $\hat{\mathbf{M_r}}$ are of rank r, and have the same norm.

The key benefit of SVD. Traffic matrices obtained from large cloud projects can have tens of thousand of rows and columns. The distance functions (used to compute row-similarity) scale exponentially in the number of features/columns. Moreover, in high dimensional feature spaces and with noisy data, distance metrics are unreliable [60] (the curse of dimensionality). Given this, a reduced-rank estimation of $\mathbf{M}$, and projection on to a feature-space of reduced dimensions allows our algorithm to remain robust to scale while retaining much of its structure.

3.4 Hierarchical Clustering

Infeasible clustering methods. Given the original matrix $\mathbf{Y}$, or the rank-r estimate $\hat{\mathbf{M_r}}$, we can use traditional clus-

tering techniques to find VM Groupings. For instance, prior approaches like [28] have established links between dimensionality reduction and K-Means clustering. However, for our use-case we would like to use dimensionality reduction for robustness to scale and noise but maintain fine control over the number of clusters to produce. Therefore, given that we do not know the number of clusters to produce, much of the existing work around K-Means [41] does not suffice for our needs. Density-based approaches such as DBSCAN [32] and OPTICS [22] do not require the number of clusters as input. However, they rely on other threshold parameters, estimating which requires domain knowledge (*e.g.,* information about a project beyond the sampled traffic volumes we have available) and maybe hard with high-dimensional data. MeanShift [29] and Affinity Propagation [33] also don't require the number of clusters, but their main drawback is time complexity, which depends on the number of iterations until convergence. We also show that MeanShift and Affinity Propagation don't perform well in the context of VM clustering in section 4.4.

Agglomerative clustering. Similar to density-based approaches, hierarchical clustering does not require the number of clusters *a priori*. CloudCluster uses agglomerative clustering [48], a bottom-up hierarchical clustering approach: each VM (row) starts in its own cluster, and clusters are recursively merged together. It uses Ward linkage [56] to determine which two clusters should be merged: at each iteration, this technique selects two clusters that minimize the increase in total within-cluster sum of squared error [44]. In the context of clustering VMs, doing this produces clusters of VMs with homogeneous traffic patterns, and this variance-minimizing property is similar to the K-Means objective function. The output of agglomerative clustering is a dendrogram (tree) of VMs (rows); the leaf nodes are the VMs (rows) and the non-leaf nodes are the nested clusters. Each non-leaf node has a value ("height"), which is the Ward distance [44] between the two entities merging at that node.

From hierarchical to flat clustering. In a dendrogram, each non-leaf node represents a potential cluster (containing all the leaf nodes in its sub-tree). CloudCluster must extract disjoint clusters from this dendrogram. To do this, it can use a static height threshold: each non-leaf node higher than this threshold is a distinct cluster. But, determining the threshold requires domain knowledge for each project. Instead, CloudCluster cuts the dendrogram based on *cluster inconsistency* [54]. For a given non-leaf node in the dendrogram with height h, if its sub-tree contains nodes with heights $H = \{h_0, h_1, ...\}$, and mean of the heights is $\overline{H}$, and the standard deviation is $\sigma(H)$, the *inconsistency* of the node is: $inc = \frac{h - \overline{H}}{\sigma(H)}$.

When deciding whether to merge two sub-trees (or nested clusters), the inconsistency metric quantifies how different the new merged cluster would be compared to the nested clusters within it. A low value means that the merged cluster would be similar to the nested clusters under it. Conversely, a high
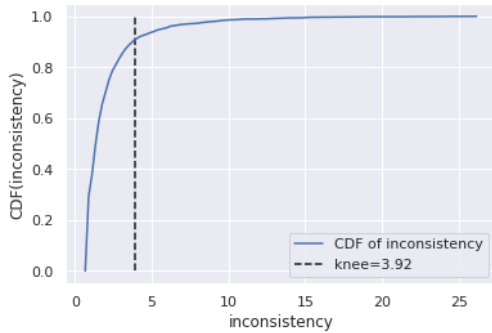
Figure 3: CDF of inconsistency value and the knee

inconsistency means that the merged cluster contains nested clusters which are fairly different. Therefore, the algorithm merges nested clusters when the inconsistency score is less than a threshold, μ.

Estimating μ. Instead of manually selecting the threshold μ, we use the following technique to estimate it. Closer to the leaves of the dendrogram, inconsistency values will be small. They will increase at non-leaf nodes higher in the dendrogram. For many projects, the distribution of inconsistency values is similar to Figure 3. This suggests that the *knee* of this curve is a good choice for μ because it identifies a transition between low and high inconsistency values. We use the knee locator implemented by [51] to determine μ. Then, we cut the dendrogram based on the threshold μ, resulting in a set of clusters.

3.5 Cluster Merging

In practice, we have found that our approach produces, for projects with thousands of VMs, tens or hundreds of clusters with small internal variation in terms of VM traffic patterns. However, it is too aggressive, and we find we can merge some of these clusters in a fast post-processing step. For this, we determine the centroid of each cluster produced by hierarchical clustering. Each centroid can be viewed as a *feature* of the candidate clusters. We treat each of these centroids as a new entity and cluster these entities. Inspired by MeanShift [29] which fuses clusters that are close to each other by comparing the distances to a threshold, we calculate the pairwise cosine distances of the clusters centroids and recursively merge pairs of clusters until no two clusters have a centroid distance less than a fixed merging threshold θ.

4 CloudCluster Evaluation

The goal of the evaluation is to demonstrate that CloudCluster produces clusters that are consistent with VMs grouped by location and function. In other words, *in each cluster, all VMs are in the same zone, and perform the same function.*

4.1 Methodology and Metrics

Dataset. We use anonymized, aggregated flow logs (specifically, Google VPC logs [10], please see footnote on page 1 for

a statement of the ethical use of customer data.) from cloud customers to generate our evaluation dataset. Our dataset includes projects ranging from a few thousand VMs to those with tens of thousands of VMs. We do not consider smaller (10-20 VMs) projects in our analysis; at these scales, less sophisticated tools can provide actionable insights. The dataset includes projects of VMs with various type of workloads (*e.g.,* web servers, load balancers, image transcoders, key-value stores *etc.*). It includes projects that are internal to Google and those belonging to external customers. Each traffic matrix in the dataset contains uniformly sampled VM-to-VM traffic aggregated over a 1-hour window. We use sampled data; sampling is necessary to scale measurement systems, and, as long as the sampling mechanism is uniform, we expect our clustering algorithm to work just as well as it would have on un-sampled data given that uniform sampling ought to preserve traffic volume relationships between VMs.

Implementation. Customer flow logs are stored in Google's Colossus file system [30]. CloudCluster loads the flow logs into Dremel [43] and uses Dremel's SQL-like queries to *select* data within the aggregation window, *group by* src-dst VM pairs and *aggregate* by volume. CloudCluster runs on a single VM with 128G memory, loads the aggregated result from Dremel into a dataframe, extracts the VM-to-VM traffic matrix, and then runs the algorithm described in §3. Traffic matrices for the projects we evaluate fit comfortably into a single VM.

Methodology. To demonstrate that CloudCluster produces clusters consistent with VMs grouped by location and function, we conduct two experiments on disjoint sets of the fifteen projects in our dataset:

i. Carefully-Named Group. The first experiment uses data belonging to eight projects. These eight projects (called the *Carefully-Named Group*) are different from the other seven projects *because we have information about the location and function of each VM*. For these projects, the customers have carefully named each VM based on function, likely to simplify manageability of the project. For example, VM naming schemes contain strings identifying well-known services (e.g. "redis" [25], "cassandra" [39], or "nginx" [53]). We call these strings *VM labels* (in addition to labels, VM names may contain, for example, instance identifiers). For projects in this group, we show that CloudCluster's clusters, when further sub-grouped by the VM location (the VM's zone, §2) match well with VM groupings by location and VM labels, *i.e.,* functions.

ii. Coarsely-Named Group. The second experiment uses data belonging to the remaining seven projects. For these, we have location information for each VM, but the VM naming scheme does not always indicate the function, or indicates function coarsely (we explain later precisely what this means). For this group of projects, we show that CloudCluster's clusters, when further sub-grouped by the VM location, *do not*

match well with VM groupings by location (zone) and VM labels.

We emphasize that the cloud provider will always know a VM's location, but cannot always know the VM's function, since function-based naming is not a requirement of any cloud-service API that we are aware of.

Metrics. We use two standard measures of clustering goodness, *homogeneity* and *completeness* [49]. These are both scalar real-valued metrics in the range $[0, 1]$. In the context of VM clustering, *homogeneity* is the fraction of VMs in a same cluster that have the same location and VM label. Conversely, *completeness* is the fraction of VMs that have the same location and VM label that are in a single cluster. These are the analogs of precision and recall used in classification.

4.2 The Carefully-Named Group

The Carefully-Named Group refers to the eight projects where the VM's are carefully named to reflect their function, in addition to the location (zone) information.

High homogeneity and completeness. We cluster the VMs in each of the eight projects in the Carefully-Named Group. As noted earlier, we further sub-group the clusters produced by location, *i.e.,* zone. Table 1's third and fourth columns show the homogeneity and completeness for all the projects in this group. Across these projects, CloudCluster has high homogeneity: all projects have a homogeneity of 0.92 or higher, and for six of them the score is higher than 0.96. Completeness scores are also high: all projects have a completeness of 0.94 or higher. High completeness and homogeneity scores indicate good matching in the clustering results, and substantiate our central assertion: that CloudCluster's clusters, when augmented with zone information, match VMs grouped by location and function.

What values of homogeneity and completeness are acceptable? Recall that these measures are the equivalent of precision and recall (respectively), for which acceptable thresholds depend upon the specific use case. Similarly, acceptable values of homogeneity and completeness depend upon what clustering is used for; we discuss this in §5.5. Also, as with precision and recall, we can trade-off homogeneity for completeness and vice versa; see §4.4 for an example.

CloudCluster works well for projects at different scales. Projects range in size from 500 VMs to over 10,000 VMs (second column of Table 1). The number of clusters (third column of Table 1) varies from a handful to around 200. CloudCluster also discovers clusters at different scales. Within project A, some clusters have more than 900 VMs, and some clusters have dozens of VMs or sometimes one. Moreover, CloudCluster can handle projects with varying functional and geographical diversity. Projects A, B, E and F each run more than 20 different kinds of software and span across a number of zones across the globe. This also explains why they have many clusters (recall that clusters are distinguished both by function and location). Projects C and D are functionally

homogeneous and scoped to a single continent; and projects G and H are moderately functionally diverse (5-6 different types of functions) but scoped within North America. This explains why C, D, G and H have only a handful of clusters.

Why location is important. Clustering groups VMs with a similar traffic pattern. Our hypothesis was that VMs that perform the same function will have similar traffic patterns. However, consider two VMs that perform the same function, but are located in zones on different continents. Although their traffic distributions to other VMs will be similar, they will likely send traffic to completely different sets of VMs (*e.g.,* load-balancers, other services) because they are located in different zones. Thus, their *rows* in the traffic matrix will be different, and CloudCluster will be unable to cluster them.

To illustrate the importance of location, for projects in the Carefully-Named Group, we compare completeness and homogeneity scores *without using location information*. This means that we no longer sub-group CloudCluster's clusters by location (zone) *and* we do not use the location information when computing homogeneity and completeness scores. Table 1's 5th and 6th columns show that, in this case, while homogeneity is reasonably high (all VMs in a cluster tend to have the same label, *i.e., function*), completeness drops significantly for about half of the projects (*i.e.,* VMs with the same label do not all fall into the same cluster).

Label and cluster conflicts. Prior work has also explored a different way to characterize the quality of clustering [46]. Consider any pair of VMs. These VMs can either be in different clusters, or they can be in the same cluster. If clustering is perfect, then (a) if the VMs belong to different clusters, they must have different labels,[7] and (b) if they belong to the same cluster, they must have the same labels. Conversely, clustering can fail in two ways: (a) the VMs belong to different clusters, but they have the same label (we call this a ***cluster conflict***, which results in a completeness score lower than 1.0) and (b) the VMs belong to the same cluster, but have different labels (we call this a ***label conflict***, which results in a homogeneity score less than 1.0).

To understand the magnitude of these conflicts, Table 1's 7th and 8th column show the *rate* of cluster and label conflicts in each of our projects in the Carefully-Named Group. Following [46], we compute the rate of cluster conflicts as the fraction of all VM pairs in different clusters that have the same label, and the rate of label conflicts as the fraction of VM pairs in each cluster that have different labels. Table 1's 7th and 8th column show that these numbers are negligibly small (less than half a percent) across all projects, and represents another way of viewing the results in Table 1's 3rd and 4th column. For instance, for project *D*, label conflicts are zero, so its homogeneity is 1. Similarly, project *C* has high homogeneity because its label conflict rate is very small and

---

[7]More precisely, different labels or locations; we use labels to simplify the explanation

Project	#VM	#Cluster	CloudCluster w/ location info		CloudCluster w/o location info		Percentage of Conflict	
			Homogeneity	Completeness	Homogeneity	Completeness	Cluster conflict	Label conflict
A	10000+	72	0.984	0.966	0.942	0.312	0.020%	0.197%
B	5000+	206	0.919	0.951	0.888	0.706	0.046%	0.532%
C	500+	4	0.999	0.964	0.989	0.865	0.001%	0.614%
D	500+	3	1.000	0.938	0.989	0.831	0.000%	0.427%
E	5000+	60	0.966	0.940	0.929	0.901	0.212%	0.386%
F	5000+	177	0.937	0.949	0.873	0.929	0.127%	0.188%
G	5000+	8	0.996	0.971	0.992	0.971	0.001%	0.169%
H	1000+	6	0.997	0.997	0.997	0.997	0.000%	0.028%

Table 1: Homogeneity and completeness score (with and without location information) and percentage of conflict for projects in the Carefully-Named Group

project *H* has highest completeness and the lowest cluster conflict rate. (As an aside, these rates are defined on VM-pairs, so the actual rates cannot directly be matched to imperfections in homogeneity and completeness.).

Why CloudCluster is less than perfect. Given the diversity of project in the Carefully-Named Group, CloudCluster's agreement with customer-provided functional groupings is impressive. However, it is less than perfect for several reasons.

Feature scaling compresses the range of each feature, which changes the relative feature distances of all VMs. Dimensionality reduction step removes information from all feature vectors. TruncatedSVD [34] only keeps the information of the specified number of dimensions. Merging might also induce errors. We use an approach similar to MeanShift's postprocessing [29] in that we merge clusters that are similar to each others by a specified distance. Even though we choose a rather aggressive merging threshold, it is still possible to merge two groups of VMs that have different traffic patterns. Similarly, the merging threshold can also be so high that it breaks other sub-clusters which should be merged.

Finally, some of these label and cluster conflicts can be caused by inconsistently assigned VM labels. For instance, in project *F*, which has high homogeneity and completeness, we found some VMs labeled default or pool. Table 1 suggests that mis-naming of VMs in our Carefully-Named Group is small. As we discuss in §4.3, CloudCluster works less well for our Coarsely-Named Group because VM naming does not reflect function (*i.e.,* from the perspective of this analysis, the VMs are mis-labeled). Equally important, the non-zero rate of label and cluster conflicts suggests that, even for well-named projects, labels may be mis-configured; in §5.3 we discuss techniques to detect such misconfigurations.

4.3 Coarsely-Named Group

In §4.2, we showed that (a) CloudCluster has high homogeneity and completeness for projects where labels reflect functions, and VM location is taken into account, and (b) it has high homogeneity and low completeness when VM location is omitted. The Coarsely-Named Group contains projects where VM labels do not reflect function well. We ex-

(a) Cluster by customer labels   **(b)** Predicted clusters
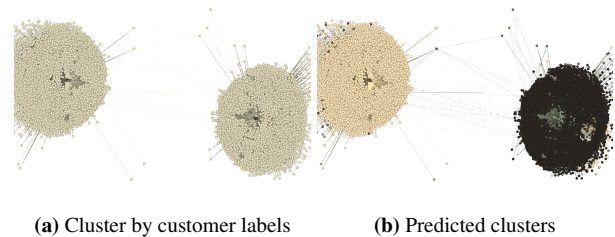
Figure 4: Same set of VMs clustered by (a) the customer label and (b) our algorithm. This figure shows VMs with generic labels like "default", "pool" or "farm".

pect CloudCluster to perform poorly in this case; we use this group of projects to rule out the possibility of other factors contributing to high completeness and homogeneity for the Carefully-Named Group.

As Table 2 shows, for projects I through O (which have comparable functional, size and spatial diversity as projects A-H), homogeneity is high, but completeness is low (for most projects in the 0.6-0.8 range, but in one case as low as 0.25). These results indicate that, in these projects VM labeling has the following property: if two VMs are similar in traffic characteristics, they are likely to have the same labels, but if they are different by traffic characteristics (so are in different clusters) they may still have the same labels. In other words, labels in these projects lack functional *specificity*.

Labeling specificity. Table 2 suggests that, if VM grouping by labels and location should match well with CloudCluster, labels have to have functional *specificity*. Some projects have less specific (or generic) functional labels, as we illustrate in the following examples.

Project	Homogeneity	Completeness
I	0.988	0.825
J	0.988	0.740
K	0.952	0.782
L	0.936	0.786
M	0.978	0.250
N	0.983	0.758
O	0.993	0.603

Table 2: Homogeneity and completeness scores for projects in the Coarsely-Named Group.

Figure 4 shows a group of VMs where the VM labels are generic (`default`, `pool`, or `farm`). In Figure 4(a), the dots are colored by the VM group they belong to by customer label. In this case, all VMs belong to the same group because they are assigned a generic label, so in Figure 4(a) they all have the same color (green). However, if we closely examine the functional structure of this project, we see two distinct groups densely connected internally, but sparsely connected externally. Figure 4(b) shows that CloudCluster is able to correctly distinguish between the two groups (yellow and black).

Sometimes, making labels specific enough requires careful thought. Figure 5 illustrates a case where *sharding* may require generic labels. In Figure 5(a), the customer has labeled all nodes within the circled ellipses as "loadbalancer". However, from Figure 5(b), we observe that there is an internal structure to these load-balancers. They can be further separated into three groups where each has a distinctive connection pattern. The clustering algorithm captures this difference and puts them into different clusters.

Customers are not required to provide specific functional labels, but these examples give some insight into how Cloud-Cluster's clustering might differ from a customer's notion of functional labels. At the same time, many projects *do* label VMs by function. For these, being able to identify generic customer labeling (or, more generally, mis-labeling) can help identify configuration errors (see §5).

4.4 Impact of Design Choices

We now quantify the importance of various design choices.

Dimensionality reduction. Dimensionality reduction reduces the runtime of the pipeline and improves the clustering accuracy. In the absence of dimensionality reduction, the distance metric can be unreliable for data with high-dimensional feature spaces [60]. Moreover, distance computation does not scale well for projects with more than 10,000 VMs; on our largest project, without dimensionality reduction, the pipeline takes more than **40 minutes** to finish. With dimensionality reduction, CloudCluster's pipeline completed in **150 seconds** for the same project. CloudCluster is not latency-sensitive, but lower computational complexity is important for reducing the overhead or cost of executing CloudCluster's algorithms

(a) Cluster by customer labels   **(b)** Predicted clusters
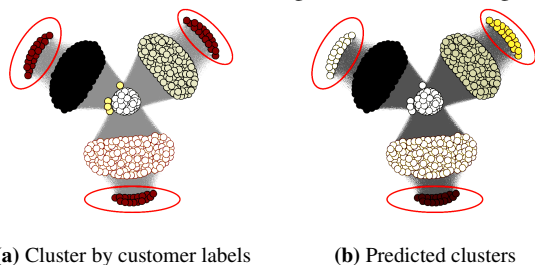
Figure 5: Same set of VMs clustered by (a) the customer and (b) our algorithm. This figure shows that customer-defined VM groups contains customer specific sharding.

on the cloud.

Feature scaling. Feature scaling approaches influence Cloud-Cluster's performance. If we disable feature scaling, Cloud-Cluster produces lower homogeneity (0.812) and completeness (0.822) scores for project A, our largest project. By contrast, log-scaling is able to achieve 0.984 homogeneity and 0.966 completeness. Using other forms of feature scaling result in slightly lower homogeneity and completeness scores. Figure 3 shows that using standardized and minmax scaling reduces both homogeneity and completeness.

Hierarchical clustering. To validate our choice of our clustering algorithm, we compare with other plausible clustering approaches. We used OPTICS [22], Affinity Propagation [33] and MeanShift [29] to produce another set of clusters, and compared the clusters with project A's labels. To be fair to these alternative clustering approaches, we performed the same feature scaling and dimensionality reduction before feeding the data into the algorithms. We used default parameters for these other clustering algorithms. We show that OPTICS results in significantly lower homogeneity (0.471) and completeness (0.163). Affinity Propagation produces slightly better homogeneity (0.994) by producing more than 3000 clusters in the project with 10,000+ VMs. This comes at the cost of a significantly lower completeness (0.559). Conversely, Mean-Shift achieves a higher completeness score (0.989) by having giant, noisy clusters, but with lower homogeneity (0.701).

Merging. Without merging, we achieve a slightly better homogeneity score (0.996), but a much worse completeness score (0.547). The high homogeneity is due to the fact that we produce clusters with small internal variation in the process of hierarchical clustering. The requirement of small internal variation divides VMs with similar traffic patterns into different clusters and lowers the completeness score. Merging combines similar clusters to significantly improve completeness for project *A* (from 0.547 to 0.966) at the expense of a small drop in homogeneity (from 0.996 to 0.984). This benefit of merging is evident across all projects in the Carefully-Named Group (Table 4). In some cases, the improvements in completeness are even more dramatic, increasing from 0.442 to 0.996 for project *H*.

	Homog.	Compl.
CloudCluster	0.984	0.966
Without feature scaling	0.812	0.822
Feature scaling: standardizer	0.939	0.953
Feature scaling: minmax scaler	0.974	0.948
Clustering: OPTICS [22]	0.471	0.163
Clustering: Affinity Prop [33]	0.994	0.559
Clustering: MeanShift [29]	0.701	0.989
Disable merging	0.996	0.547

Table 3: Compares the impact of different design choice on project A's result

	Without Merging		With Merging	
	Homog.	**Compl.**	**Homog.**	**Compl.**
A	0.996	0.547	0.984	0.966
B	0.932	0.896	0.919	0.950
C	0.998	0.522	0.998	0.963
D	1.000	0.442	1.000	0.938
E	0.985	0.698	0.965	0.940
F	0.978	0.781	0.937	0.949
G	0.996	0.386	0.996	0.971
H	0.997	0.442	0.996	0.996

Table 4: Effect of merging on the Carefully-Named Group group

5 CloudCluster For Project Management

In this section, we describe several proof-of-concept ways in which the output of CloudCluster can help cloud providers provide their customers with actionable insights about the configuration and management of their services.

5.1 Reconfiguration to Reduce Cost

Cloud providers often price traffic in multiple tiers: traffic within the same cloud zone typically costs less than traffic between VMs from different zones, regions or continents. Customers engineer VM placements to reduce cost while balancing availability and proximity to customers. Cloud-Cluster can help identify opportunities for reconfiguring VM placements to reduce costs. In this section, we discuss three examples that illustrate these opportunities; future work can develop systematic tools to discover such opportunities.

Figure 6 shows the distribution of traffic to other zones from VMs of project A belonging to VM label L. CloudCluster detects that VMs with this label belong to two different clusters: one which sends traffic more-or-less uniformly to VMs in 8 different zones (first cluster in Figure 6) and the other which sends over 80% of its traffic to a single zone (second cluster). A customer can potentially reconfigure the placement of VMs of the latter cluster to avoid inter-zone traffic. Although the traffic skew is visible across all VMs (so the customer might have been able to detect it using the VM label), CloudCluster is able to identify the precise set of VMs to re-configure.

Using CloudCluster, the cloud provider can determine the volume of intra-cluster and inter-cluster traffic, and determine how much of this traffic crosses zone, region, or continent boundaries. Using this, it can estimate cost savings resulting from reconfigured VM placements. Figure 7 and Figure 8 illustrate cost savings from reconfiguration in two cases.

The first case is a cluster C from project A of VMs located in different zones of a single cloud region. Almost 90% of traffic in C is intra-zone, which is free or relatively cheap on most cloud providers ([15], [8], [11]). However, the remaining traffic traverses continental boundaries, and accounts for a significant fraction of total cost charged to C. If the customer were to provision a small cluster in the zone on the other continent where the traffic comes from, it can reduce the cost attributable to this cluster by 41.2% (Figure 7).

The second case is a cluster C' of project A whose traffic is largely inter-region (intra-zone traffic is $< 0.1\%$). 92.3% of egress traffic from C' goes to zones in another region R, and 95.9% of its ingress traffic comes from VMs in a single zone in R. Moving VMs in C' to R (an *egress-favored* placement) reduces cost by 21.1%, while moving these VMs to the zone in R from which they receive most traffic (an *ingress-favored* placement) reduces cost by 15.1% (Figure 8).

These are simplified examples; in practice, tools that suggest re-configuration of VM placements will need to consider other customer objectives such as availability and latency. We have left development of such tools to future work, but Cloud-Cluster's clustering can be a valuable input to such tools.

5.2 Anomaly Detection

Large-scale cloud project outages are sometimes caused by rapid increases in service workload, management operations by the customer (incorrect service configuration), by the provider (VM migration), or failures in the provider's network. These are often accompanied by sudden shifts in traffic between VMs in the service or traffic to and from external entities (*e.g.,* customers of the cloud service). Such traffic shifts may often be visible in the aggregate traffic between clusters. Because our clusters correspond to functionally homogeneous VMs, if one VM in cluster A starts communicating more with a VM in cluster B, it is likely that all other VMs in A will also start communicating more with VMs in B.

In this section, we present a preliminary evaluation of an anomaly detector that tracks significant deviations in aggregate inter-cluster traffic on each link in the inter-cluster graph (Figure 1(e)). Such a detector can also help localize anomalies, as we discuss below. In practice, we expect our anomaly detector to complement other approaches used by cloud providers.

The anomaly detector works as follows. For each edge in the inter-cluster graph (an edge exists between two clusters if their VMs communicate), it tracks at each aggregation window, the total volume in bytes, the total flow count, and the number of communicating VM pairs between each pair of clusters. When, for a given edge, any of these quantities deviates significantly from a windowed moving median [57], we flag that deviation as an anomaly (we omit the details for brevity). Because the inter-cluster graph is sparser than the inter-VM graph (*e.g.,* Figure 1(a)), we are able to scalably identify correlated anomalies, where two or more communicating cluster pairs exhibit anomalous traffic at the same time.

Trace Analysis and Results. To quantify the effectiveness of this detector, we identified 25 time windows across different projects where either (a) an internal anomaly detector that uses a different methodology flagged anomalous traffic in the project during the corresponding time window (17 instances) or (b) the customer filed a trouble ticket (8 instances).

We then ran the CloudCluster-based anomaly detector on these 25 time windows, and, in each case, were able to con-
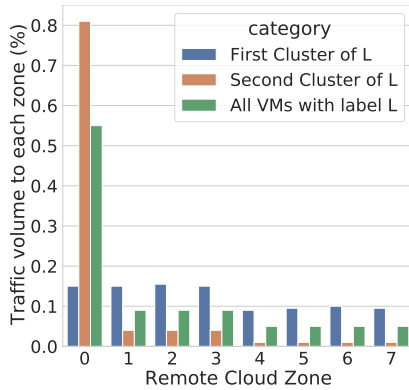
Figure 6: CloudCluster finds VMs with same label but different traffic patterns.
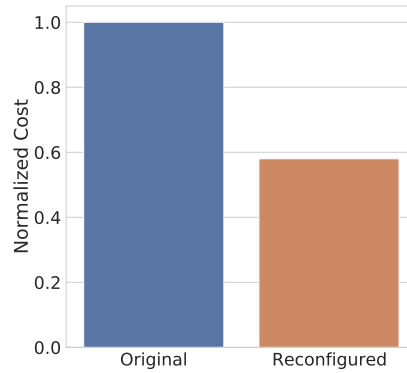
Figure 7: Original vs. Reconfigured placement cost
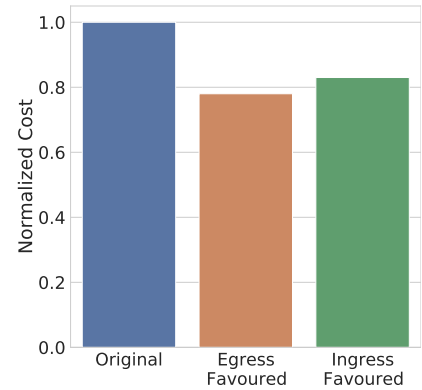
Figure 8: Original vs. Egress-favored vs. Ingress-favored placement cost

firm the existence of the anomaly[8], and also to pinpoint which cluster-pairs were responsible for the deviations. We have not analyzed false positive rates; since we started with known anomalies flagged by other systems. For the 8 customer-reported incidents, our detector was able to correctly identify the offending cluster-pairs (as determined in the post-mortem reports). We identified two broad classes of anomalies: *traffic shifts* and *structural changes*. In the first class, the inter-cluster graph does not change, but traffic on some subset of links changes significantly. In the second, new nodes and or edges are added to the graph or nodes and edges are removed. Of the 25, three were traffic shifts and the rest were structural changes.[9]

Our detector is fast: the maximum processing latency to compute the deviation scores, across all projects, was 92.3 milliseconds per time window.

The following paragraphs briefly describe some qualitatively different anomalies that we were able to detect; §A contains a more detailed description.

Correlated traffic shift due to peering router failure. This anomaly was reported by the network operator in reaction to a peering router failure. Our detector observed that a cluster in the region nearest the peering router saw a sudden reduction in flow and byte counts. Concurrently, a cluster in another region, (which, from label names, we determined was functionally identical to the first cluster), saw an increase in traffic. We suspect that the peering router failure diverted external traffic to enter the cloud provider's network at a different location, but don't have the instrumentation to confirm this.

Structural change due to VM migration. This anomaly was reported by the internal anomaly detector. Our CloudCluster-based anomaly detector identified a sequence of structural changes across successive aggregation windows. Recall that clusters are distinguished both by function and lo-

cation (§4.2). In this case, the structural changes were caused by a migration of VMs from one server to another due to scheduler-driven evictions. The migration was spread out over multiple aggregation windows, so our detector noticed a sequence of structural changes corresponding to progressive migration of VMs from one server to another.

Structural change due to project reconfiguration. Our internal anomaly detector flagged anomalous traffic for a cloud provider. The CloudCluster-based anomaly detector identified a structural change: two clusters were removed from the graph and one was added. The two initial clusters corresponded to a singleton cluster containing a leader VM and another containing 120 worker VMs. The new cluster contained the 121 VMs, encompassing both the leader and the workers. In this case, it turns out that the customer had initiated the structural change, decommissioning the older VMs in favor of another set of VMs as part of an upgrade.

5.3 Potential Label Misconfiguration

As discussed in §4.2, several customers label VMs with precise function names and location information. We conjecture that they use this to simplify project management. These VM labels are often configured, either by hand or by a script. *Label misconfigurations* can occur, and CloudCluster can be used to detect the *likely* candidates. When a label misconfiguration occurs in a project whose VMs appear to be named by function and location, *i.e.,* when the project has a high homogeneity and completeness, it manifests either as a label conflict or a cluster conflict (§4.2).

Cluster Conflict. A cluster conflict occurs when VMs belong to different clusters, but have the same labels. Such a conflict can either result from a misconfigured label, or from a clustering error. To distinguish between those two cases, we use a technique inspired by prior work in clustering on *silhouette analysis* [50], which attempts to measure the intrinsic performance of clustering. This analysis assigns each item (or VM, in our context) a score in the range $[-1, 1]$ that measures how similar the VM is to its own cluster, compared to other

---

[8]A more detailed analysis of the detector performance, and comparisons with other detection techniques, is beyond the scope of this paper.

[9]Some of these structural changes or traffic shifts might be intentional, even though our approach flags them as (statistical) anomalies.

clusters.

We modified this idea to derive a metric that quantifies whether a customer label is too generic (*i.e.,* spans multiple clusters) or too specific. Let V_l be the set of VMs that has a customer-defined label l, but CloudCluster splits it up into n clusters $\{C_1, C_2, ...C_n\}$. Let $\bar{c}_l$ be the centroid, in feature space, of the traffic features of all VMs in V_l. Let $\bar{c}_i$ be the centroid of the traffic features of all VMs in C_i (the C_is might contain VMs not in V_l). For each cluster C_i, let a_i be the average distance of each VM in this cluster to $\bar{c}_l$ and b_i be the average distance of each VM in this cluster to $\bar{c}_i$. Then, consider the following metric: $ms(i) = \frac{b(i)-a(i)}{max(a(i),b(i))}$.

Intuitively, if $ms(i) < 0$, each VM in the cluster is closer to the cluster center than to the label's center, so the labeling is too generic. Conversely, if $ms(i) > 0$, then the label is too specific. Either way, this indicates a mismatch between clustering and customer-provided labeling, which can be used in some cases to identify potential mis-labeled VMs.

To detect mis-labeling VMs using this technique, we apply the following algorithm. Without loss of generality, assume that C_1 has the largest number of VMs with label l. For all $i > 1$, if $ms(i) < \psi$ (a conservative threshold < 0, we use -0.5), we mark all VMs in C_i with label l as mis-named.

The output of this analysis is a list of *potentially* (we use this term to indicate that, ultimately, any such mis-labeling would have be be verified by a customer, since the customer understands the *intent* behind the naming) mis-labeled VMs that cause cluster conflicts.

Table 5 lists the fraction of potentially mis-labeled VMs for four of our projects. These four projects belonged to a customer who gave us feedback on our clustering results. The fraction of mis-labeled VMs range from negligible amounts (*e.g.,* project *H* has 0.1% mis-named VMs) to a few percent (for projects *A*, *E* and *F*). For these projects, we were able to verify with the customer that our identification of mis-labeled VMs was accurate. In these cases, the customer had changed the functions in some VMs but forgot to update the VM labels.

Label conflicts. Mis-labeling can also cause label conflicts: different labels within the same cluster. Table 5 also shows the rate of occurrence of these. They happen less frequently, and often fall into two categories. VMs labeled generically such as "default" fall into the same cluster as VMs with more specific labels (*e.g.,* "app-server"). A second cause of mis-labeling is inconsistent hyphenation (*e.g.,* "appserver" vs. "app-server"), or inconsistent abbreviations (*e.g.,* using "es" instead of "east"). We identified examples in the second category using manual inspection; future work can automate the detection of mis-labeling in this category using edit-distance based string similarity analysis [55].

5.4 Potentially Mis-provisioned VMs

When configuring a VM, project owners can *provision* VM resources by specifying the *machine type* for each VM. Machine types determine the capacity of the VM instances

Project	Cluster Conflict (%)	Label Conflict (%)	Mis-provisioning Rate (%)
A	4.62	0	1.59
E	5.75	3.15	0
F	7.26	0.10	0.80
H	0.09	0.04	0.38

Table 5: The percentage of VMs that are mis-labeled in each project (§5.3), the rate of misprovisioning (§5.4).

in terms of CPU cores, memory and egress network bandwidth [35]. Different machine types are priced differently, so over-provisioning a VM can have cost implications. Misprovisioning can also impact performance: under-provisioned VMs can result in stragglers, causing services to violate their latency SLOs.

CloudCluster can identify mis-provisioned VMs by determining outlier machine types in a cluster. Since CloudCluster's clusters identify VMs performing a similar function, if most VMs in a cluster are of machine type *a*, but a small number are of machine type *b*, we can identify the latter set as mis-labeled VMs. In determining the rate of mis-labeling, we must filter out mis-labeled VMs. To be more robust to clustering errors, we flag a VM as mis-labeled if it does not lie at the edge of the cluster (as determined by distance from the cluster centroid in feature space).

Table 5 shows the rate of mis-provisioning in 4 of the projects in the Carefully-Named Group. A small number, 1%, appear to be mis-provisioned. We say "appear to be" because the operator cannot know the intent of the customer; they may have deliberately provisioned these machines differently to run additional tasks (*e.g.,* compute bound jobs whose footprint is not visible in the VM-to-VM traffic matrix). Any misprovisioning will ultimately have to be verified as such by manual inspection by the customer.

5.5 Discussion

In §4.2, we said that acceptable values of homogeneity and completeness depend upon what clustering is used for. We conclude this section with a brief qualitative discussion of this issue, leaving quantitative analysis to future work.

We have described two types of use cases in this section. Reconfiguration and anomaly detection are based upon the inter-cluster graph abstraction, and specifically upon inter-cluster traffic volumes. For these cases, if *most* VMs (*e.g.,* 90%) in a cluster are functionally similar, the reconfiguration decision, or the anomaly detection is likely to be correct.

Detecting mis-labeled or mis-provisioned VMs requires comparing attributes of VMs within a cluster. This can be more susceptible to false positives and false negatives, unless homogeneity and completeness are very high. Because a cloud provider cannot always know the homogeneity and completeness a priori, using clustering for these tasks requires additional filtering steps to minimize false positives. For misprovisioned VMs, we filter candidates at the edge of the cluster (§5.4). For mis-labeling, we use silhouette analysis (§5.3).

6   Related Work

Inferring Structure from Traffic. Complementary to Cloud-Cluster, others have explored inferring host behavior and distributed system properties from network traffic. The closest prior work [58] groups Internet hosts within each IP prefix by traffic similarity, and explores how this can be used to detect malicious behavior. Other work has modeled host-to-host communication as a graph to understand properties of inter-host communication [13, 14, 36], to infer botnet structure [45], or the logical structure of enterprise networks [16]. The body of work on tracing in distributed systems seeks to infer the causal structure as well as other properties of distributed systems from RPC traces to aid performance debugging (*e.g.,* [19, 47, 52]). Other work has used traffic to infer specific characteristics of VMs in cloud settings: strongly connected groups of VMs as candidates for migration [26], or compromised VMs [23]. Some of these use clustering [26, 58], but do not consider scale and robustness to range of traffic volumes.

Data Clustering. Clustering is a mature area of research, with many established techniques such as K-Means [41], DB-SCAN [32], OPTICS [22], AffinityPropogations [33], Hierarchical Clustering [48], *etc.* That clustering is susceptible to the curse of dimensionality is well-known [60]. Clustering in high dimensions has been explored extensively either by: (a) using heuristics to determine attributes of sub-spaces (*e.g.,* CLIQUE [18] or SUBCLU [37]) or (b) designing special distance measures (*e.g.,* projected clustering, as in Pre-DeCon [24] or PROCLUS [17]). In contrast, CloudCluster explicitly reduces the dimension of the VM-to-VM traffic to the point where conventional clustering techniques and similarity measures are applicable.

Cloud Monitoring and Workload Characterization. Tangentially relevant prior work has used CPU and memory utilization traces to infer properties of VMs [31, 38, 42].

7   Conclusion

CloudCluster performs clustering on the VM-to-VM traffic of cloud projects and yields the functional structure of the cloud service. It overcomes the challenges imposed by scale (cloud services contain tens of thousands of VMs), by orders-of-magnitude variability in traffic volume and measurement noise, and by the lack of prior knowledge of the cloud projects (for number of clusters). The output of CloudCluster can help detect potentially mis-provisioned or mis-labeled VMs, identify opportunities to reduce cost, and detect anomalies.

Future work. Several directions of future work remain, including: identifying the frequency at which to apply Cloud-Cluster to projects; incrementally adjusting clusters when VMs leave or join; supporting traffic to cloud native services such as storage; exploring better methods for determining the cluster merging threshold; more thoroughly evaluating the accuracy of cost reconfiguration, anomaly detection, or miscon-figuration determination, and comparing their performance against other alternatives; determining whether additional information from customers, obtained with their consent, can improve the quality of the resulting functional structure.

References

[1] 6.3. preprocessing data. https://scikit-learn.org/stable/modules/preprocessing.html#preprocessing.

[2] Azure monitor overview - azure monitor. https://docs.microsoft.com/en-us/azure/azure-monitor/overview.

[3] Cloud monitoring | google cloud. https://cloud.google.com/monitoring.

[4] Creating and managing projects. https://cloud.google.com/resource-manager/docs/creating-managing-projects.

[5] Google cloud storage. https://cloud.google.com/storage.

[6] Google vpc. https://cloud.google.com/vpc/docs.

[7] Network and subnetwork terminology. https://cloud.google.com/vpc/docs/vpc#subnets_vs_subnetworks.

[8] Network pricing|compute engine documentation|google cloud. https://cloud.google.com/compute/network-pricing.

[9] Regions and zones | compute engine documentation | google cloud. https://cloud.google.com/compute/docs/regions-zones.

[10] Using vpc flow logs. https://cloud.google.com/vpc/docs/using-flow-logs.

[11] Virtual network pricing: Microsoft azure. https://azure.microsoft.com/en-us/pricing/details/virtual-network/.

[12] Clouds project cloudwatch. https://aws.amazon.com/cloudwatch/, 2000.

[13] Blinc. *ACM SIGCOMM Computer Communication Review*, 35(4):229–240, 2005.

[14] Network monitoring using traffic dispersion graphs (TDGs). *Proceedings of the ACM SIGCOMM Internet Measurement Conference, IMC*, (c):315–320, 2007.

[15] Aws site-to-site vpn and accelerated site-to-site vpn connection pricing. https://aws.amazon.com/vpn/pricing/, 2020.

[16] Role classification of hosts within enterprise networks based on connection patterns. *Proceedings of the General Track: 2003 USENIX Annual Technical Conference*, pages 15–28, 2020.

[17] Charu C Aggarwal, Joel L Wolf, Philip S Yu, Cecilia Procopiuc, and Jong Soo Park. Fast algorithms for projected clustering. *ACM SIGMoD Record*, 28(2):61–72, 1999.

[18] Rakesh Agrawal, Johannes Gehrke, Dimitrios Gunopulos, and Prabhakar Raghavan. Automatic subspace clustering of high dimensional data. *Data Mining and Knowledge Discovery*, 11(1):5–33, 2005.

[19] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, page 74–89, New York, NY, USA, 2003. Association for Computing Machinery.

[20] Muhammad Amjad, Vishal Misra, Devavrat Shah, and Dennis Shen. Mrsc: Multi-dimensional robust synthetic control. *Proc. ACM Meas. Anal. Comput. Syst.*, 3(2), June 2019.

[21] Muhammad Amjad, Devavrat Shah, and Dennis Shen. Robust synthetic control. *Journal of Machine Learning Research*, 19(22):1–51, 2018.

[22] Mihael Ankerst, Markus M Breunig, Hans-Peter Kriegel, and Jörg Sander. Optics: ordering points to identify the clustering structure. In *ACM Sigmod record*, volume 28, pages 49–60. ACM, 1999.

[23] Behnaz Arzani, Selim Ciraci, Stefan Saroiu, Alec Wolman, Jack Stokes, Geoff Outhred, and Lechao Diwu. Privateeye: Scalable and privacy-preserving compromise detection in the cloud. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 797–815, Santa Clara, CA, February 2020. USENIX Association.

[24] Christian Bohm, K Railing, H-P Kriegel, and Peer Kroger. Density connected clustering with local subspace preferences. In *Fourth IEEE International Conference on Data Mining (ICDM'04)*, pages 27–34. IEEE, 2004.

[25] Josiah L. Carlson. *Redis in Action*. Manning Publications Co., USA, 2013.

[26] Marco Cello, Kang Xi, Jonathan H Chao, and Mario Marchese. Traffic-aware clustering and vm migration in distributed data center. In *Proceedings of the 2014 ACM SIGCOMM workshop on Distributed cloud computing*, pages 41–42, 2014.

[27] Sourav Chatterjee. Matrix estimation by universal singular value thresholding. *The Annals of Statistics*, 43(1):177–214, Feb 2015.

[28] Michael B. Cohen, Sam Elder, Cameron Musco, Christopher Musco, and Madalina Persu. Dimensionality reduction for k-means clustering and low rank approximation. In *Proceedings of the Forty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '15, page 163–172, New York, NY, USA, 2015. Association for Computing Machinery.

[29] D. Comaniciu and P. Meer. Mean shift: a robust approach toward feature space analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(5):603–619, May 2002.

[30] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Dale Woodford, Yasushi Saito, Christopher Taylor, Michal Szymaniak, and Ruth Wang. Spanner: Google's globally-distributed database. In *OSDI*, 2012.

[31] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 153–167, New York, NY, USA, 2017. Association for Computing Machinery.

[32] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, page 226–231. AAAI Press, 1996.

[33] Brendan J. Frey and Delbert Dueck. Clustering by passing messages between data points. *Science*, 315(5814):972–976, 2007.

[34] Nathan Halko, Per-Gunnar Martinsson, and Joel A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions, 2009.

[35] Google Inc. Machine types | compute engine documentation | google cloud. https://cloud.google.com/compute/docs/machine-types.

[36] Yu Jin, Esam Sharafuddin, and Zhi-Li Zhang. Unveiling core network-wide communication patterns through application traffic activity graph decomposition. *SIGMETRICS Perform. Eval. Rev.*, 37(1):49–60, June 2009.

[37] Karin Kailing, Hans-Peter Kriegel, and Peer Kröger. Density-connected subspace clustering for high-dimensional data. In *Proceedings of the 2004 SIAM international conference on data mining*, pages 246–256. SIAM, 2004.

[38] Arijit Khan, Xifeng Yan, Shu Tao, and Nikos Anerousis. Workload characterization and prediction in the cloud: A multiple time series approach. In *2012 IEEE Network Operations and Management Symposium*, pages 1287–1294. IEEE, 2012.

[39] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[40] Christophe Leys, Christophe Ley, Olivier Klein, Philippe Bernard, and Laurent Licata. Detecting outliers: Do not use standard deviation around the mean, use absolute deviation around the median. *Journal of Experimental Social Psychology*, 49(4):764–766, 2013.

[41] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA, 1967.

[42] Shruti Mahambre, Purushottam Kulkarni, Umesh Bellur, Girish Chafle, and Deepak Deshpande. Workload characterization for capacity planning and performance management in iaas cloud. In *2012 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, pages 1–7. IEEE, 2012.

[43] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: Interactive analysis of web-scale datasets. *Proc. VLDB Endow.*, 3(1–2):330–339, September 2010.

[44] Fionn Murtagh and Pierre Legendre. Ward's hierarchical clustering method: clustering criterion and agglomerative algorithm. *arXiv preprint arXiv:1111.6285*, 2011.

[45] Shishir Nagaraja, Prateek Mittal, Chi-Yao Hong, Matthew Caesar, and Nikita Borisov. Botgrep: Finding p2p bots with structured graph analysis. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, page 7, USA, 2010. USENIX Association.

[46] William M. Rand. Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical Association*, 66(336):846–850, 1971.

[47] Patrick Reynolds, Janet L. Wiener, Jeffrey C. Mogul, Marcos K. Aguilera, and Amin Vahdat. WAP5: Black-box performance debugging for wide-area systems. *Proceedings of the 15th International Conference on World Wide Web*, pages 347–356, 2006.

[48] Lior Rokach and Oded Maimon. *Clustering Methods*, pages 321–352. Springer US, Boston, MA, 2005.

[49] Andrew Rosenberg and Julia Hirschberg. V-measure: A conditional entropy-based external cluster evaluation measure. In *Proceedings of the 2007 joint conference on empirical methods in natural language processing and computational natural language learning (EMNLP-CoNLL)*, pages 410–420, 2007.

[50] Peter J. Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20:53 – 65, 1987.

[51] Ville Satopaa, Jeannie Albrecht, David Irwin, and Barath Raghavan. Finding a" kneedle" in a haystack: Detecting knee points in system behavior. In *2011 31st international conference on distributed computing systems workshops*, pages 166–171. IEEE, 2011.

[52] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[53] Igor Sysoev et al. Nginx. *Inc.,"nginx," https://www. nginx. com*, 2004.

[54] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake Vand erPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1. 0 Contributors. SciPy 1.0–Fundamental Algorithms for Scientific Computing in Python. *arXiv e-prints*, page arXiv:1907.10121, Jul 2019.

[55] Robert A Wagner and Michael J Fischer. The string-to-string correction problem. *Journal of the ACM (JACM)*, 21(1):168–173, 1974.

[56] Joe H. Ward. Hierarchical grouping to optimize an objective function. *Journal of the American Statistical Association*, 58(301):236–244, 1963.

[57] Eric W Weisstein. Moving median. https://mathworld.wolfram.com/MovingMedian.html.

[58] Kuai Xu, Feng Wang, and Lin Gu. Network-aware behavior clustering of internet end hosts. In *2011 Proceedings IEEE INFOCOM*, pages 2078–2086. IEEE, 2011.

[59] Y. Zhang and Z. Ge. Finding critical traffic matrices. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 188–197, 2005.

[60] Arthur Zimek, Erich Schubert, and Hans-Peter Kriegel. A survey on unsupervised outlier detection in high-dimensional numerical data. *Statistical Analysis and Data Mining: The ASA Data Science Journal*, 5(5):363–387, 2012.
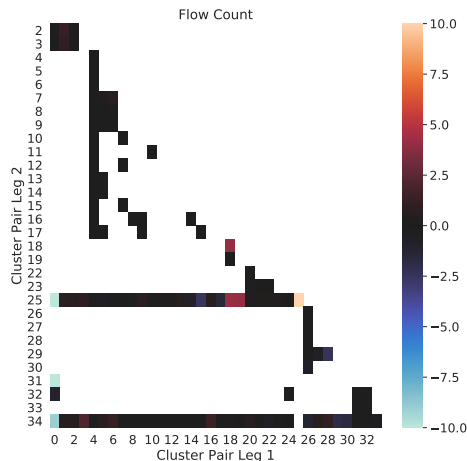
Figure 9: Cluster to Cluster flow count deviation scores

Figure 10: Regional anomaly scores during VM eviction/migration

Appendix

A Detailed Explanation of Anomalies

Correlated traffic shift due to peering router failure. This anomaly was reported by the network operator in reaction to a peering router failure. Our detector observed that a cluster in the region nearest the peering router saw a sudden reduction in flow and byte counts. Concurrently, a cluster in another region, (which, from label names, we determined was functionally identical to the first cluster), saw an increase in traffic. We suspect that the peering router failure diverted external traffic to enter the cloud provider's network at a different location, but don't have the instrumentation to confirm this. Figure 9 depicts the cluster to cluster deviations scores expressed in terms of median absolute deviations (MAD) [40] with the sign indicating whether the upper (e.g. positive) or lower (e.g. negative) anomaly detection boundary was crossed. The x-axis and y-axis represent either the source or destination of the cluster pairs whose interactions are studied. The values of the heat map show the MAD deviations observed between the interacting cluster pair and is assigned a color based on the color map where the color black indicates no deviation and warmer (calmer) colors represent anomalous traffic characteristic deviating above (below) the median traffic volume. In the observed traffic shift, we were quickly able to identify the cluster pairs impacted by the anomaly (e.g. $|dev| > 5$ in Figure 9) that appear as the red, orange, and blue cells in the heatmap, filter away clusters not impacted by the anomaly that appear as black cells in the heatmap, and provide a project wide summary of the anomaly.

Structural change due to VM migrations. This anomaly was reported by the internal anomaly detector. Our CloudCluster-based anomaly detector identified a sequence of structural changes across successive aggregation window. Recall that clusters are distinguished both by function and location (§4.2). In this case, the structural changes were caused by a migration of VMs from one server to another due to scheduler-driven evictions. The migration was spread out
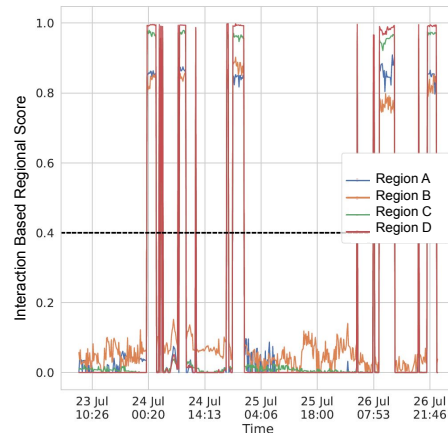
over multiple aggregation window, so our detector noticed a sequence of structural changes corresponding to progressive migration of VMs from one server to another. Figure 10, shows the regional-score, an average of all the cluster to cluster deviation scores weighted by the number of VM communication pairs, computed for every region. The recurring structural changes manifest as plateaus and valleys in the regional score observed. The valleys represent the time windows where the new cluster behaviour is learnt and clusters behave as *normal*, while the plateau's illustrate the time windows where there is a migration storm (e.g. significant number of VM migrations).

Load increase. A customer reported, to the cloud provider, a trouble ticket asking to root cause a missed service-level agreement. The CloudCluster-based anomaly detector identified a sudden increase of external traffic to clusters with memcached servers in one of the customer's projects. (CloudCluster models external traffic as a single node in the inter-cluster graph). This was also concluded in the manual postmortem of the event. Similar to Figure 9 where we characterize the deviation in cluster pair interactions, here we observed a drift in interactions against logical clusters (e.g. may not contain VMs such as client IP) representing connections external to the project. Using this, we identified the culprit traffic flows through heavy hitter analysis and their origin, which happens to be a project that auto scaled to keep up with traffic demand and subsequently flooded the memcached projects with requests.

Structural change due to project reconfiguration. Our internal anomaly detector flagged anomalous traffic for a cloud provider. The CloudCluster-based anomaly detector identified a structural change: two clusters were removed from the graph and one was added. The two initial clusters corresponded to a singleton cluster containing a master VM and another containing 120 worker VMs. The new cluster contained the 121 VMs, encompassing both the master and the workers. In this case, it turns out that the customer had initiated the structural
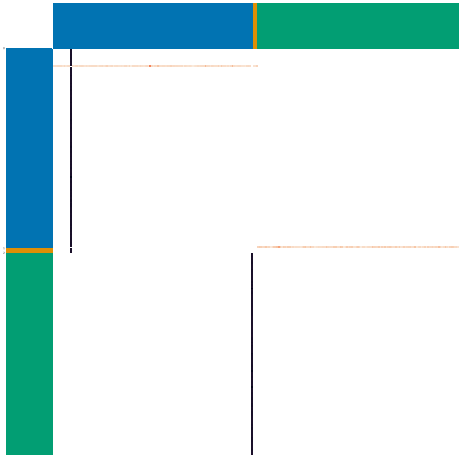
Figure 11: VMxVM Communication Graph (grouped by cluster assignment)

change, decommissioning the older VMs in favor of another set of VMs as part of an upgrade. Figure 11 shows the project communication structure using a VMxVM matrix where the values of the heatmap show the number of bytes sent between VMs log scaled (e.g. white represents no communication and warmer colors depict higher bandwidth consumption). The VMs in the rows and columns are sorted by their cluster assignments wherein they are grouped and identified by the color strip (e.g. cluster id) that appear on the top and left side of the heatmap (e.g. blue, yellow and green). On visualizing the project structure using this heatmap, the aforementioned project re-instantiation evolves in the following manner: a) Initially the top-left sub-structure (e.g. one master, 120 workers) operated devoid of the bottom-right substructure. b) After a downtime where the cluster-to-cluster deviation scores exceed a predefined threshold, we observed the bottom-right structure, which had displaced the other sub-structure. Therefore, by analyzing the traffic patterns, the network-engineer can identify changes to the project structure.

Zeta: A Scalable and Robust East-West Communication Framework in Large-Scale Clouds

Qianyu Zhang[1], Gongming Zhao[1*], Hongli Xu[1*], Zhuolong Yu[2], Liguang Xie[3]

Yangming Zhao[1], Chunming Qiao[4], Ying Xiong[3], Liusheng Huang[1]

[1]*University of Science and Technology of China,*
[2]*Johns Hopkins University,* [3]*Futurewei Technologies,* [4]*SUNY at Buffalo*

Abstract

With the broad deployment of distributed applications on clouds, the dominant volume of traffic in cloud networks traverses in an east-west direction, flowing from server to server within a data center. Existing communication solutions are tightly coupled with either the control plane (*e.g.*, preprogrammed model) or the location of compute nodes (*e.g.*, conventional gateway model). The tight coupling makes it challenging to adapt to rapid network expansion, respond to network anomalies (*e.g.*, burst traffic and device failures), and maintain low latency for east-west traffic.

To address this issue, we design Zeta, a scalable and robust east-west communication framework with gateway clusters in large-scale clouds. Zeta abstracts the traffic forwarding capability as a Gateway Cluster Layer, decoupled from the logic of control plane and the location of compute nodes. Specifically, Zeta adopts gateway clusters to support large-scale networks and cope with burst traffic. Moreover, a transparent Multi IPs Migration is proposed to quickly recover the system/devices from unpredictable failures. We implement Zeta based on eXpress Data Path (XDP) and evaluate its scalability and robustness through comprehensive experiments with up to 100k container instances. Our evaluation shows that Zeta reduces the 99% RTT by 5.1× in burst video traffic, and speeds up the gateway recovery by 10.8× compared with the state-of-the-art solutions.

1 Introduction

With an increasing number of distributed applications (*e.g.*, MapReduce [82] and Elasticsearch [32]) on the clouds, east-west communication between instances has become the majority load (even up to 75% [17]) in cloud networks [65]. In addition, cloud providers usually offer isolation for tenants through Virtual Private Cloud (VPC) [77]. Therefore, it is essential for cloud networks to support high-speed and reliable intra-VPC communication [83]. However, two factors

bring much pressure on cloud networks. On one hand, a large-scale cloud can accommodate over 100k servers and millions of instances with Pbps bandwidth [7], bringing congestion risks to the network. According to the monitoring log of a cloud with 1,500 servers, we can observe congestions that last over 1s for more than 12,500 times in one day [38]. On the other hand, containerization leads to centralized startup and short life cycles of instances, which bring great dynamics to the network. For example, Google launches several billion containers per week into Google Cloud [31, 50].

As a result, the east-west communication between instances faces several challenges in large-scale and highly dynamic cloud networks. (1) *Scalability.* The expansion of the instances scale in cloud networks leads to a rapid increase in forwarding rules consumption. For example, the control plane will install 487M rules for a preprogrammed network with 40k instances [22], which brings high latency on the rules lookup and traffic forwarding. Therefore, installment of numerous rules will limit the size of a single VPC and the whole network. (2) *Robustness.* Although the failure probability of a specific equipment is usually low, network abnormal events in large-scale clouds are frequent and inevitable, including device failures [18, 59] and burst traffic [68, 81]. They pose severe network congestion/interruption and degrade the tenants' experience. (3) *Latency.* The latency of configuring forwarding rules and establishing/resuming communication is a crucial metric. When instances launch/migrate, some previous solutions require the control plane to inform all relevant hosts and install/update rules, which especially affects short-lived tasks. For example, a function task (*e.g.*, MilliSort and MilliQuery [47]) usually completes in milliseconds, while it may take a few seconds to launch a function instance and establish connection for it.

The existing east-west communication solutions in cloud networks are usually divided into two main categories. One is the hardware solutions, such as AWS Nitro System [6, 67], Azure FPGA-based SmartNIC [28, 46, 61] and AliCloud P4-based Gateway [57]. The other is the software solutions, including the preprogrammed model (*e.g.*, VMware NSX
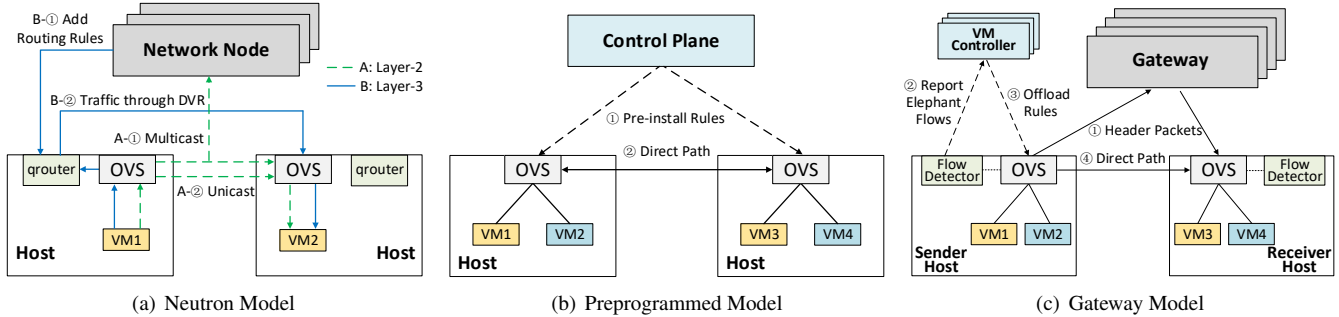
Figure 1: Three Typical East-West Communication Models. (a) Neutron model realizes layer-2 communication by learning MAC address and utilizes DVR (qrouters) for layer-3 communication. (b) Preprogrammed model pre-installs all potential rules when launching VMs. (c) Gateway model pre-installs default rules pointing to the gateway on the host. The gateway forwards the header packets, and the direct path rules of elephant flows will be offloaded to the source hosts.

[39, 56]) and the gateway model (*e.g.*, Google Cloud Hoverboard [22]). Considering the high cost and long development cycles of hardware, software solutions have become the preferred choice for many medium-sized cloud providers. However, the existing software solutions also face several critical disadvantages (see §2.1 for details). First, the preprogrammed model pre-installs numerous rules for VMs and is coupled with the control plane. The conventional gateway model depends on fixed gateways allocated for host zones and is coupled with the location of compute nodes. Hence, they lack the scalability or robustness to adapt to large-scale networks. Second, the existing control loops are complex, which aggravates the recovery delay in network abnormal events, including device failure/overload and VM migration.

To overcome the above challenges, we propose a scalable and robust east-west communication framework in large-scale clouds, called Zeta. Zeta abstracts the traffic forwarding capability as a gateway cluster layer, decoupled from the location and logic of other modules. Specifically, Zeta mainly proposes the following innovative designs. (1) Zeta utilizes gateway clusters to improve the fault tolerance of a single gateway and leverages eXpress Data Path (XDP) [36] to accelerate gateway forwarding, thereby enhancing the network scalability and robustness. (2) Zeta adopts the flow table and group table [84] to realize the intra-cluster gateway load balancing. (3) Zeta proposes Multi IPs Migration to achieve gateway fast recovery, which implements failover by migrating the vIPs of the failed gateways. This scheme avoids updating the on-host default rules pointing to the gateways, making failure recovery transparent to hosts/tenants.

The main contributions of this paper are as follows:

- We analyze the pros and cons of existing typical east-west communication models in large-scale clouds and present the design principles for our framework.

- We design a prototype framework, called Zeta, to achieve scalable and robust east-west communication in large-scale clouds. Zeta is publicly available at https://github.com/futurewei-cloud/zeta/.

- We evaluate the robustness and scalability of Zeta through comprehensive experiments. Evaluation results show that Zeta reduces the 99% RTT by 5.1× in burst video traffic, and speeds up the gateway recovery by 10.8× compared with the state-of-the-art solutions.

2 Background and Motivation

We will analyze the limitations of three typical east-west communication models in large-scale clouds and motivate our work in this section.

2.1 Limitations of Prior Works

As an open source cloud computing architecture, OpenStack helps quickly deploy small-scale clouds [63]. As shown in Figure 1(a), OpenStack Neutron provides the networking capability for the clouds. Specifically, Neutron provides layer-2 networking communication by learning MAC address [55]. When two VMs in the same layer-2 domain communicate for the first time, the source VM will broadcast ARP packets to obtain the MAC address of the destination VM. However, when encountering burst traffic in large-scale networks, it may cause unnecessary layer-2 broadcasts and unicast flooding, leading to poor robustness and scalability [71]. For layer-3 networking, all the traffic will be routed by specific network node(s) in the initial OpenStack releases. It may suffer the risk of network node(s) failure and high forwarding delay in large-scale networks. To this end, OpenStack has released the Distributed Virtual Router (DVR) since Juno version [13], which can significantly mitigate the robustness and latency issues. However, DVR suffers the oversize routing tables and frequent synchronization problems, which also decrease the network scalability [64]. In general, OpenStack gradually improves forwarding performance through evolutions. But due to the lack of targeted designs for large-scale clouds, it still faces robustness and scalability issues.

Table 1: Comparison of the advantages and disadvantages of existing models

Models	Robustness		Scalability		Latency		
	Gateway Failure	Burst Traffic	VPC Size	Global Scale	Forwarding	VM Launching	VM Migration
Neutron [13, 55]	✗	✗	✗	✗	✗	✓	✗
Preprogrammed [39]	—	✓	✗	✗	✓	✗	✗
Gateway [22]	✗	✗	✓	✓	✓	✓	✓
Ours: Zeta	✓	✓	✓	✓	✓	✓	✓

To reduce the forwarding latency between VMs, the preprogrammed model was adopted by many early platforms, such as VMware NSX [39, 56]. As shown in Figure 1(b), the control plane pre-installs all potential rules when launching VMs, as it cannot exactly predict which pairs of VMs will communicate. The traffic between VMs will be forwarded directly with low delays. However, the preprogrammed model brings some nonnegligible system overhead. First, it will pre-install a quadratic number of rules on hosts, which limits the network scalability. Specifically, in a cloud network with h hosts and n VMs, $2n$ rules should be pre-installed before launching a new VM in the worst case, and there will be O($n \times h$) rules in the system. A massive number of pre-installed rules will slow down the rules lookup and traffic forwarding, thus limiting the network scale. Second, numerous preprogrammed rules seriously delay the VMs deployment/migration. The control plane needs to pre-install/update all potential rules on hosts, which will cause a significant delay in communication establishment/recovery. For example, the preprogrammed model takes 74 seconds to install 487M rules for a large network with 10k hosts and 40k VMs [22, 39]. Above system overhead leads to poor scalability and flexibility of the preprogrammed model, especially in large-scale cloud networks.

To overcome the disadvantages of the former two models, the gateway model on-demand installs rules, and has been widely adopted by cloud providers, such as Google Cloud Hoverboard [22]. As shown in Figure 1(c), the gateway model organizes all servers into host zones. Host zone/cluster is a collection of colocated machines with uniform network connectivity, each of which is equipped with a master gateway and several backups. This model only pre-installs default rules pointing to the gateway on the host's vSwitch. When a new flow arrives, the vSwitch sends the header packets to the gateway according to the default rules. Then, the gateway forwards these packets and offloads direct path rules for elephant flows [22], so that the subsequent packets of those elephant flows will be forwarded to the destination directly.

The gateway model improves the network scalability through on-demand rules offloading. However, it allocates a fixed number of gateway(s) to each host zone and may encounter the robustness issues. 1) *Gateway failure.* Although the master-backup gateway model provides disaster tolerance, it will take a long time to migrate all the traffic from the mas-

ter gateway to the backup ones, and cannot effectively cope with gateway failures. For example, it takes 260-310ms [72] to inform 14 affected hosts and update the default entries on each OVS. The recovery delay far exceeds the carrier-grade requirements of 50ms [52, 72, 78]. The network interruption caused by the excessive recovery delay will seriously decrease the QoS. 2) *Burst traffic.* The gateway model only assigns a master gateway to each host zone. When a host zone encounters burst traffic, the corresponding master gateway will be easily overloaded (especially when the control plane cannot detect and offload high bandwidth flows immediately).

2.2 Our Intuitions

As summarized in Table 1, the gateway model combines the advantages of both Neutron and Preprogrammed model in terms of scalability and latency. However, the existing gateway model usually assigns fixed gateway(s) to each host zone. Its gateways incur a high risk of overload/failure under abnormal events, including burst traffic and gateway failures. A natural solution is to deploy multiple master gateways in a host zone to alleviate the impact of burst traffic or abnormal events. However, the gateways need to be provisioned for peak bandwidth usage, making it difficult to efficiently schedule gateway resources.

Another intuitive solution is to organize all gateways into a large virtual cluster to improve disaster tolerance. The new arrival flows will be forwarded to gateways through ECMP [69]. However, once VMs launching/migration occurs, the control plane should notify all gateways to update the forwarding rules, which brings high synchronization overhead on both the gateways and the control plane [60]. For example, assuming that a large datacenter contains 500 gateways and launches 3k containers per second [31, 50]. The controller needs to send 1.5M update messages in one second, which poses a severe risk of control plane overload. Obviously, this solution is not feasible for large-scale clouds.

In order to integrate the pros, but mitigate the cons of models discussed above, we divide all gateways into multiple clusters. A gateway cluster can effectively improve fault tolerance while reducing the synchronization overhead, as the controller only needs to push latest forwarding rules to the gateways of one cluster every time. Moreover, we abstract the gateways'

forwarding capability as Gateway Cluster Layer, which is decoupled from the location and logic of other planes/modules. On the one hand, we utilize gateway clusters independent of the compute nodes to enhance robustness and achieve high performance. We adopt the Multi IPs scheme, which is transparent to hosts/tenants to achieve gateway fast recovery. The independence of gateway cluster gives us flexibility in building high-performance data plane. We choose XDP as the data plane of Zeta, because of its integration with Linux kernel and similar speed as DPDK [24]. On the other hand, the new framework allows easy integration with existing cloud platforms. Thus, we can make full advantage of existing designs, such as Open vSwitch (OVS) [58] group table. According to the above ideas, we design a scalable and robust east-west communication framework in large-scale clouds to support high-performance traffic forwarding.

3 System Design

3.1 Design Goals

Zeta is an east-west communication framework with gateway clusters in large-scale clouds. Our design goals are as follows:

- **Robustness:** High reliability is the core requirement of east-west communication, especially for cloud providers. In particular, Zeta focuses on effectively dealing with burst traffic and abnormal events (*e.g.*, gateway failure/overload/expansion), to avoid network congestion/interruption degrading the tenants' experience.

- **Low Latency:** Since east-west traffic is very sensitive to latency. Zeta aims to reduce the latency of the traffic forwarding through the high-performance in-kernel fast-path. In addition, the lightweight control loop helps reduce the delay of VMs launching/migration.

- **Scalability:** With the rapid growth of cloud scale, Zeta should better support large-scale virtual networks up to 100k instances.

- **Compatibility:** Zeta is open source and can also serve as a common hosting platform to integrate customization network functions into the overall virtual networking.

3.2 System Overview

As shown in Figure 2, to realize the above design goals, we propose an efficient east-west communication framework, called Zeta, which consists of three core modules: Gateway Cluster, On-host Forwarding and Framework Management.

Gateway Cluster Layer establishes a forwarding network based on VXLAN tunnel [48]. It leverages XDP to provide high-performance traffic forwarding and on-demand rules offloading for tenant instances (§4.2). The application
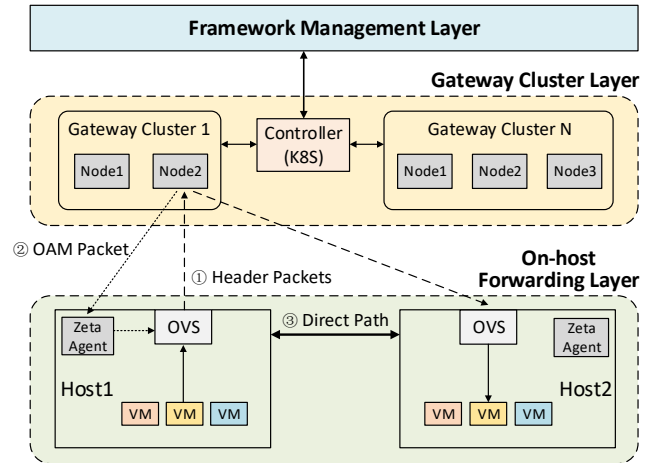
Figure 2: Zeta Framework Overview. Gateway Cluster provides high-performance traffic forwarding and on-demand rules offloading for tenant instances. On-host Forwarding transmits traffic according to default/direct rules and achieves the intra-cluster gateway load balancing through group tables. Framework Management manages the whole network and further improves the system robustness through scheduling.

of gateway cluster ensures better scalability and robustness. Gateways detect the elephant flows and sends OAM (Operations, Administration and Maintenance) packets to the source hosts, which contain direct path rules (§4.3). In addition, Zeta adopts *Multi IPs Migration* to achieve fast recovery from gateway failure/overload/expansion, which makes failure recovery transparent to hosts/tenants (§4.4).

On-host Forwarding Layer transmits traffic according to the rules on OVS. Before deploying a new VPC, a default rule will be pre-installed on the host, which consists of a flow entry and a group entry to achieve the intra-cluster gateway load balancing (§5.1). When two VMs communicate for the first time, the header packets will be sent to a specific gateway according to the default rule. Each host deploys a *Zeta Agent*, which is responsible for parsing OAM packets and installing the direct path rules on the on-host OVS. In addition, the lightweight control loop based on *Zeta Agent* can make a quick response to network adjustments, such as passive instance migration (§5.2).

Framework Management Layer manages the whole network and further enhances the robustness of gateway clusters. When Zeta is initialized, the management layer will determine the VPC-cluster mapping for inter-cluster load balancing (§6.1). To deal with the abnormal events and traffic dynamics, the *Multi IPs Scheduler* will dynamically adjust the configurations (*e.g.*, multi IPs allocation and cluster partition), thereby avoiding overload of partial clusters for better robustness (§6.2).

4   Gateway Cluster Design

4.1   Gateway Cluster Overview

Zeta Gateway Cluster establishes a VXLAN-based forwarding network. Specifically, it provides high-performance traffic forwarding and on-demand rules offloading for tenant instances with scalability and robustness guarantee. As shown in the left plot of Figure 3, Gateway Cluster Layer consists of a cluster controller and several gateway clusters.

Cluster Controller contains management and scheduling logic for gateway clusters. On the one hand, it facilitates the interaction with the Framework Management Layer through its Northbound RESTful API, such as receiving forwarding rules. On the other hand, it manages the gateway clusters and maintains the gateways load balancing through its Southbound API based on gRPC [66]. Cluster Controller is deployed within its own Kubernetes cluster hosted on Zeta control node(s).

Gateway Clusters constitute the data plane of the forwarding network. We divide all gateways into several clusters to achieve the robust gateway forwarding. In practice, each cluster consists of several isomorphic gateways, which store the same forwarding rules to collectively provide traffic forwarding and rules offloading services for tenant instances. Each gateway contains the Forwarding Module (FWD) and the Distributed Flow Table Module (DFT). Specifically, FWD forwards the packets to the destination hosts and offloads direct forwarding rules to the source hosts for those elephant flows. DFT is a lightweight key-value store, which maintains a consistent forwarding table on each gateway of a cluster. When the forwarding table changes (*e.g.*, instances launching/migration), the cluster controller will push the latest rules to each gateway of the corresponding cluster. In addition, there is no state synchronization among gateways (in §4.3).

4.2   XDP-based Traffic Forwarding

The forwarding module of a Zeta gateway is implemented based on XDP [36] to improve the forwarding performance and reduce the transmission latency. XDP is a high-performance and programmable network data path, which can directly process layer-2 frames at the NIC driver and hence bypass the kernel network stack [12, 36, 79]. As illustrated in the right plot of Figure 3, we converge the forwarding, computing and storage functions together, which eliminates the overhead of network stack processing [14, 49].

Forwarding Module works at the NIC driver and can directly operate on raw Ethernet frames. The workflows of XDP-based forwarding program are as follows: (i) Receiving header packets of the source instance from the NIC RX buffer. (ii) Obtaining the forwarding rule of the target instance by querying the storage module, that is, determining the destination host of the traffic. (iii) Parsing the protocol field of VXLAN inner packets. ARP messages will be directly re-
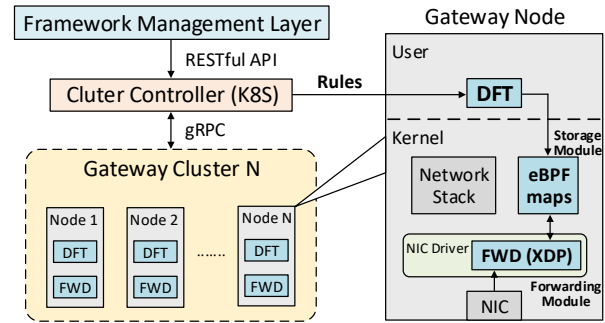
Figure 3: Illustration of Gateway Cluster Design. The left plot is the overview of gateway cluster and the right plot is the implementation details of XDP-based gateway.

sponded to the source instance, while other types of packets will be forwarded to the destination. (iv) Sending OAM (Operations, Administration and Maintenance) packets containing direct rules to the source hosts for the elephant flows.

Storage Module consists of several eBPF maps [2, 19]. These maps are key-value stores [29] that serve as the data channel between DFT and FWD. The forwarding module will also cache the real-time information of flows in eBPF maps. For example, FWD will count the OAM packets generated for each flow to avoid repeatedly offloading one flow.

4.3   Gateway Flow Detection

In order to further reduce the rules stored on the hosts, so as to conserve memory and reduce the forwarding delay caused by rules lookup. Zeta adopts XDP's high-performance packet processing features to detect elephant and mice flows on the gateway, which can improve the efficiency of the detection program and the system's robustness. When encountering burst traffic generated by a simultaneous batch of workloads (*e.g.*, MapReduce [82]), the on-host flow detection program of existing gateway model may be overloaded, as its host agent is usually equipped with limited resources, *e.g.*, 1 CPU core and 1.5GB memory [22]. In contrast, the additional overhead of detecting elephant flows is almost negligible for the XDP-based gateways of Zeta while forwarding traffic.

When traffic arrives at the XDP forwarding module, it will accumulate the total size of each flow in a certain period and store the records in an eBPF LRU Hash map [44, 79]. If the cumulative size of a flow exceeds the threshold (*e.g.*, 20kbps [22]) before the next period, it will be identified as an elephant flow and offloaded to the source hosts. Each flow is only sent to a specific gateway according to the 5-tuple hash (in §5.1), which avoids synchronization of flow size statistics among gateways. In addition, Zeta will monitor the gateway load. When a gateway's CPU or memory utilization reaches the threshold (*e.g.*, 80%), the gateway will pause the elephant flows detection and offload direct rules for all flows.

4.4 Dealing with Failures through Multi IPs

The number of gateways in a cluster will change dynamically due to gateway failures and scaling requirements, and the hash modulo of the default rule will change accordingly (*i.e.*, group entry buckets in §5.1). Therefore, we have to modify all the installed default rules associated with the updated cluster. To this end, massive affected hosts need to be informed, which leads to heavy notification overhead and unacceptable delay [54]. To address this issue, we design the *Multi IPs Migration*. Briefly, each gateway node is logically assigned multiple virtual IPs (vIPs), and the vIPs can be reallocated among nodes. Tenant traffic is bound to vIPs and decoupled from gateways.

The feature of XDP working in the layer-2 networking inspires a solution of gateway failure recovery. We propose the *Multi IPs* scheme to achieve fast failure recovery. Specifically, the cluster controller maintains a *Multi IPs Mapping Table*. When a gateway cluster is initialized, each gateway node in the cluster will be allocated several logical virtual IP-MAC pairs, and send RARP packets [27] to add MAC table entries on the connected ToR switch(es). It should be noted that these vIPs and vMACs are not actually configured in the gateways' NIC, as XDP program can directly operate on the raw Ethernet frames. When a gateway fails, the cluster controller will reassign the logical vIP-vMAC pairs of the failed gateway to other healthy gateways in the cluster. Since the forwarding rules maintained by each gateway in a cluster are consistent, there is no synchronization overhead/delay among gateways during failure recovery. Next, the healthy gateways that have obtained migrated vIP-vMAC pairs will utilize RARP to inform the connected switch(es) to update MAC address table. Then, the packets from instances can be correctly forwarded to healthy gateways.

Figure 4 illustrates an example of fast recovery through *Multi IPs Migration*. Initially, Gateway Cluster 1 contains three gateway nodes, each of which is assigned with two vIP-vMAC pairs, as shown in the *Multi IPs Mapping Table*. When Node2 fails, the cluster controller will update the mapping table, ip3-mac3 and ip4-mac4 originally assigned to the Node2 are reassigned to Node1 and Node3 respectively. Next, Node1 and Node3 utilize the RARP protocol to update the MAC address table of the connected ToR switch, so that the packets toward the failed Node2 will be immediately diverted to the healthy nodes. As a result, the failure recovery is transparent to hosts/tenants without modifying any default OVS entry or on-host ARP cache that involves the failed gateway(s). According to the experiments in §8.3.2, Zeta reduces the average gateway recovery latency from 62ms to 5.5ms.

In conclusion, the *Multi IPs Migration* scheme only needs to update the IPs mapping table and send the RARP packets to ToR switches. The recovery process does not require the participation of control plane or hosts. Therefore, the failure recovery delay and the notification overhead can be almost
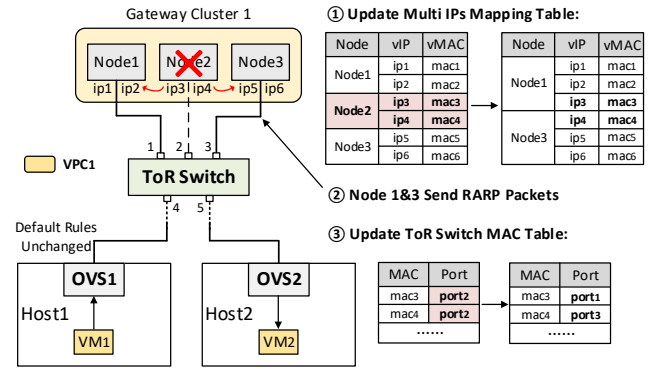
Figure 4: Dealing with Gateway Failures through *Multi IPs*. When Node2 fails, the cluster controller first updates the mapping table to reassign the vIP-vMAC pairs to healthy gateways (*i.e.* Node 1&3). Then Node 1&3 send RARP packets to update the MAC entries on the connected switch. The recovery scheme avoids modifying the default OVS rules on hosts.

negligible. It significantly enhances the robustness of gateway clusters. In addition, the *Multi IPs Migration* can also be applied in (1) Intra-cluster load adjustment and (2) Rapid cluster scaling (covered in §6.2).

5 On-host Forwarding Design

5.1 Load Balancing through Group Tables

This section elaborates on the designs of default entries to achieve intra-cluster gateway load balancing. In order to realize the decoupling of gateway cluster and location (*i.e.*, host or host zone), we construct default rules in VPC granularity. Thus, when launching a new VPC on a compute node, the default rule of this VPC will be pre-installed by Zeta Agent on the on-host OVS.

To achieve the gateway load balancing within a cluster, we utilize the flow table and group table of on-host OVS to orchestrate the gateway clusters. Specifically, each entry of the group table points to a cluster, and the buckets in each group entry specify the gateway nodes in this cluster. When the header packet of a flow reaches OVS, it first matches the flow entry and jumps to a group entry according to the VPC identifier (VPC_id) so that the target cluster for this flow is determined. The VPC-cluster mapping algorithm will be elaborated in §6.1. Then, the packet will be hashed to a bucket in the group entry, which determines the target gateway for this flow. The group entry selects the target gateway based on the 5-tuple hash of a flow. Finally, the load balancing within a gateway cluster can be guaranteed.

We give an example in Figure 5 to illustrate the intra-cluster gateway load balancing with the flow table and group table. Assuming that VM1 belonging to VPC1 communicates with VM3 for the first time. When the header packet arrives at
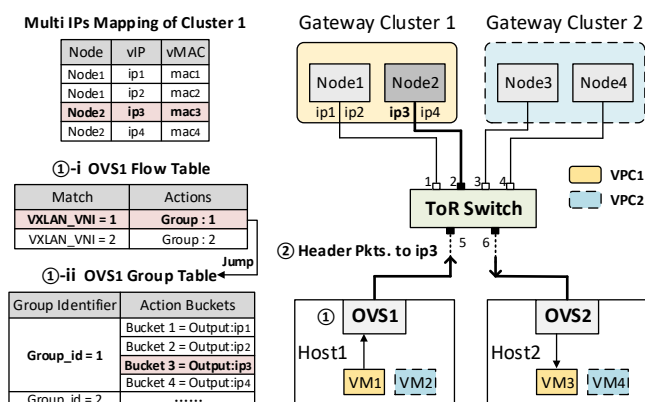
Figure 5: Illustration of Interaction between Flow Table and Group Table. When VM1 belonging to VPC1 communicates with VM3 for the first time, Host1 lookups the OVS1's default tables, and the default gateway IP of VM1's flow is ip2. Then, Host1 sends the header packets of VM1 to Node2.

the OVS of Host1, the OVS first matches the flow entry with `VXLAN VNI=1` and jumps to the group entry with `Group_id=1`. Each bucket in a group entry corresponds to the IP address of a gateway node in the cluster, and the packet will be hashed to a bucket according to its 5-tuple information. In our example, the packet is hashed to bucket3, that is, the destination address of the packet is ip3. Then, Host1 sends the header packets of VM1 to Node2, and the gateway will forward these packets and offload a direct rule to the source Host1.

5.2 Lightweight Control Agent

The lightweight control loop based on *Zeta Agent* can effectively reduce the recovery latency of the passive instance migration, such as Kubernetes Pod Eviction [42]. In a Kubernetes cluster, when a compute node is out of resources, the Kubernetes scheduler [43] will migrate the relevant pod(s) to other host(s). Conventionally, Kubernetes does not inform its networking plugin (*e.g.*, Flannel [4] and Calico [1]) of pod(s) migration actively. The networking plugin needs to poll Kubernetes database (*e.g.*, Etcd [3]) to obtain the latest pod information. Therefore, the hosts cannot update the installed direct rules immediately. The traffic is still forwarded to the former destination hosts, which results in a network interruption between the affected pods.

Three steps are required in Zeta to restore communication: (i) Obtaining the latest forwarding rules. (ii) Redirecting the packets toward the migrated pods to the correct destination. (iii) Updating the direct rules on the source hosts. We hope *Zeta Agent* remains lightweight to occupy fewer host resources. Meanwhile, Zeta gateways support the above operations. Thus, instead of directly implement above three steps on agent, the traffic towards the migrated pods will be redirected to the gateways and forwarded to the correct destinations.
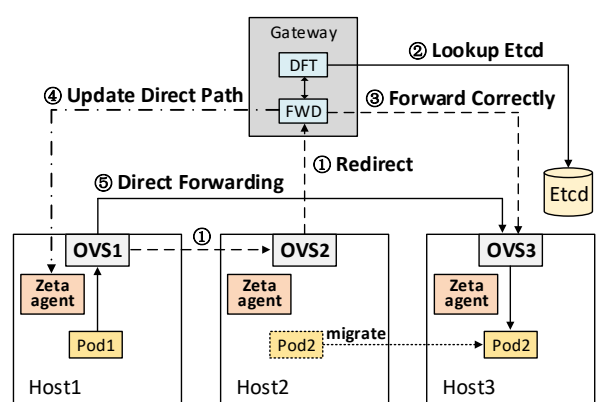
Figure 6: Lightweight Control Agent on compute nodes. When Pod2 is migrated, the flows sent to Pod2 will be redirected to gateway. The gateway forwards the flows and queries the database, then updates the direct path on the source host.

As illustrated in Figure 6, when Pod2 is migrated, the Zeta Agent on Node2 will install an entry on OVS2 to redirect all packets toward the Pod2 to the gateway. FWD on Zeta gateway recognizes the redirected packets and reports their destinations to DFT. DFT queries the latest location information of Pod2 from Kubernetes database and updates the rules cache of FWD. Then, FWD will forward the redirected packets to the correct destination Node3, and send OAM packets to the source Node1. Finally, the Zeta Agent on Node1 will update the direct forwarding rule to Pod2.

6 Framework Management Design

6.1 Gateway Cluster Mapping

When Zeta is initialized, the management layer will determine the VPC-cluster mapping for inter-cluster load balancing.

Gateway Cluster Model. In the Zeta framework, we use $C = \{c_1, c_2, ..., c_n\}$ to denote the gateway clusters, where $n = |C|$ is the number of clusters. For each gateway cluster c, its forwarding capacity is denoted as $B(c)$. We denote $V = \{v_1, v_2, ..., v_m\}$ as the VPC set, where $m = |V|$ is the number of VPCs in the cloud. Let $T = \{t_1, t_2, ..., t_{|T|}\}$ denote the tenants set and each tenant $t \in T$ consists of a VPC set $V_t = \{v_1^t, v_2^t, ..., v_{|V_t|}^t\}$. Obviously, $V = V_1 \cup V_2 ... \cup V_{|T|}$. Moreover, the traffic demand of each VPC is denoted as $f(v)$.

Problem Formalization. We define the gateway clusters mapping (GCM) problem in the Zeta framework. To enhance the system robustness and improve the QoS, we need to consider the following two constraints. (1) *VPC Constraint*. A VPC will be mapped to one and only one gateway cluster, as all the vIPs of a group entry belong to the same cluster (§5.1). (2) *Tenant Constraint*. We limit the number of gateway clusters that each tenant can be mapped to. For security reasons,

we do not expect that burst/malicious traffic from a single tenant will affect all gateway clusters.

Moreover, we use binary $x_v^c \in \{0,1\}$ to denote whether a VPC $v \in V$ is mapped to a gateway cluster $c \in C$ or not. Let binary $y_t^c \in \{0,1\}$ represent whether the gateway cluster $c \in C$ is assigned the VPCs belonging to tenant $t \in T$ or not. The objective of GCM is to achieve the load-balancing among all gateway clusters. We formulate GCM as follows:

$$\min \; \lambda$$

$$
S.t. \begin{cases}
\sum_{c \in C} x_v^c = 1, & \forall v \in V \\
\sum_{v \in V} x_v^c \cdot f(v) \leq \lambda B(c), & \forall c \in C \\
x_v^c \leq y_t^c, & \forall v \in Vt, c \in C, t \in T \\
\sum_{c \in C} y_t^c \leq k, & \forall t \in T \\
x_v^c \in \{0,1\}, & \forall v \in V, c \in C \\
y_t^c \in \{0,1\}, & \forall t \in T, c \in C
\end{cases}
\tag{1}
$$

The first set of equations means that all traffic of a VPC will be forwarded to one gateway cluster by default. The second set of inequalities describes the traffic load on each gateway cluster, where $\lambda \in [0,1]$ represents the load balancing factor. The third set of inequalities indicates that the tenant t is mapped to gateway cluster c only if VPC(s) of tenant t is processed by cluster c. The fourth set of inequalities represents the *Tenant Constraint*, that is, the VPCs of a tenant will be mapped to at most k gateway clusters. Our objective is to achieve the load balancing among all gateway clusters, *i.e.*, minimizing the load balancing factor λ.

We give an empirical formula to set the tenant constraint k in §A.1, and propose a rounding-based algorithm for the VPC-cluster mapping in §A.2.

6.2 Multi IPs Scheduler

The *Multi IPs Scheduler* executes the IPs migration scheme proposed in §4.4. It dynamically updates the IPs allocations to eliminate the overload of gateway clusters caused by the burst traffic and abnormal events. In practice, when a gateway exceeds the load threshold (*e.g.*, 80%), it will immediately report such overload to the control plane. Then the *Multi IPs Scheduler* starts to perform the following two steps:

Step 1: Intra-Cluster Load Adjustment. The scheduler first sorts all gateways of a cluster in the descending order of their load. Next, the scheduler attempts to migrate a vIP-vMAC pair from the overloaded gateway to the gateway with the lightest load, and re-sorts gateways' load. Then, the scheduler will repeat above IPs migration and gateway sorting procedure until none of the gateways in the cluster is overloaded. If we cannot eliminate the overloaded gateways with step 1, the scheduler will go to step 2.

Step 2: Cluster Scaling. If a cluster cannot eliminate overload through internal load adjustment, *e.g.*, a legitimate VPC

has burst traffic. The scheduler will migrate gateways from other clusters to this cluster or expand new gateways for this cluster. The scheduler first sorts all the clusters by their average load in the descending order and attempts to reassign a gateway from the least loaded cluster to the overloaded cluster. We can utilize *Multi IPs Migration* to achieve rapid gateway migration among clusters. However, if the gateway migration causes overload risk to the source cluster, the scheduler will directly expand the overloaded cluster with a new gateway.

7 Implementation

We implement Zeta based on Linux 5.4 kernel. The Cluster Controller includes 3k lines of Python code, the XDP-based gateway forwarding function includes 4.5k lines of C code, and the Zeta Agent includes 2k lines of C++ code.

Zeta provides two deployment methods. One is based on physical machines, and we give a best practice in §B.2. The other is based on Kernel-based VMs (KVMs), which can quickly deploy dozens of KVM-based gateways on several physical machines (see §B.3 for more details).

8 Evaluation

We first conduct an ablation analysis to measure the performance of Zeta gateway. We then test the robustness of Zeta under burst traffic and abnormal events. Finally, we evaluate the scalability of Zeta in public and private cloud scenarios.

8.1 Experimental Setting

Testbed Setups. We use 23 servers to build the testbed, all running Ubuntu 18.04 with Linux kernel 5.4. Considering our limited number of servers, we deploy KVM-based gateways on several physical machines to simulate gateway clusters. In addition, we launch a large number of container instances on each compute node to evaluate scalability, because of limited number of compute nodes. The scalability in this paper mainly refers to the instance scale, instead of the host scale, as the forwarding rules stored in the gateways and the tenant traffic depend on the instance scale.

Specifically, 20 servers are compute nodes, each equipped with dual 22-core Intel Xeon 6161 CPUs, 640GB memory and an Intel XL710 40GbE NIC. The other 3 servers are used to deploy gateway clusters, each equipped with dual 16-core Intel Xeon E5-2697A CPUs, 256GB memory and an Intel XL710 40GbE NIC. We deploy a total of 45 KVM-based gateways on the 3 physical gateway machines. Each KVM-based gateway is equipped with 4 vCPUs and 16GB memory. For Zeta, we divide the 45 gateways into 10 clusters.

Moreover, according to the empirical data in [22], we set the rules offloading threshold to 20kbps on the gateway.

Benchmarks. We compare the robustness and scalability of Zeta with other three typical frameworks. The first framework is the conventional gateway model [22], called GWZone, and its gateway is modified based on the implementations of Zeta's gateway. It allocates a master gateway for each host zone and equips backups to deal with gateway failure. Unlike Zeta, GWZone detects elephant flows on compute nodes. When GWzone faces gateway failure, it will update the default entries on affected hosts and migrate traffic to the backup gateways. We equip GWZone with 9 additional backup KVM-based gateways. As the backup gateways only consume ∼0.1 vCPU and ∼2GB memery in standby, they will not affect the performance of the master gateways. The second one is the OpenStack Neutron [55], which provides layer-2 networking communication by learning MAC address. The third one is the Preprogrammed model, which is a simplified implementation of VMWare NSX [39, 56] as it is not open source. The Preprogrammed model will pre-install all potential rules before launching VMs.
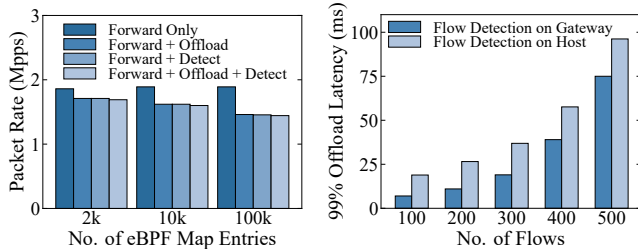
Figure 7: Packet Rate of a Physical Core vs. Entries

Figure 8: 99% Offloading Latency vs. No. of Flows

8.2 Microbenchmark

We first evaluate the impact of flow detection and rules offloading on forwarding performance with a physical core. We use iPerf [37] to generate UDP traffic, and the inner packet size is 64 bytes. In addition, the number of entries stored in eBPF map ranges from 2k to 100k. As shown in Figure 7, a single physical core can forward 1.86M packets per second under 2k entries. When the rule offloading or flow detection is supplied, the forwarding rate reduces by 8.1% to 1.71Mpps, as these two functions introduce additional eBPF map read/writes for flow statistics. After adding both flow detection and offloading functions on the gateway, the performance decreases slightly. For example, the forwarding rate only reduces by 1.2% from 1.71Mpps to 1.69Mpps under 2k entries. This is because the map read/writes are the majority overhead for forwarding, while the detection and offloading functions require the same number of map read/writes. When the number of entries scales to 100k, the forwarding rate with rule offloading and flow detection drops by 14% to 1.45Mpps, as the timeout mechanism of maps for flow statistics leads to throughput degradation with the number of entries increasing. We will optimize the timeout mechanism in future work.

We then measure the rules offloading latency with flow detection on gateway and host. The gateway still performs traffic forwarding and rules offloading with a physical core. We use iPerf to generate UDP flows on a host, each of which is 10Mbps. Figure 8 shows that flow detection on gateway can reduce the 99th percentile of offloading latency by 22% under 500 flows compared with that on host, as the performance of on-host detection is worse than XDP on gateways.

In general, flow detection on gateway can reduce the rules offloading latency (*e.g.*, reduce 22% as shown in Figure 8) and has little impact on the forwarding performance (*e.g.*, decrease 1.2% from 1.71Mpps to 1.69Mpps as shown in Figure 7). Thus, Zeta detects elephant flow on gateways for faster rules offloading with little detection overhead.

We also evaluate the linear scaling throughput of Zeta gateways (§C.2).

8.3 Robustness Evaluation

In this section, we evaluate the performance of Zeta under various burst traffic workloads and different abnormal events. Based on the further transformation (§C.1) of Google cluster-data [30], we deploy 100 VPCs with 2,000 VMs on the 20 compute nodes. Each VPC contains 10-90 VMs, and each VM is equipped with 1 vCPU and 6GB memory.

8.3.1 Robustness under Burst Traffic

We compare the robustness of the Zeta gateway cluster with GWZone under burst traffic of different applications. We choose three typical traffic workloads according to the traffic characteristics in cloud networks [8, 45], including MapReduce, video and audio. Specifically, we deploy a MapReduce cluster in each VPC and execute the word-counting application on each MapReduce cluster simultaneously with input size of 10GB, which mainly generates TCP elephant flows. We also deploy video and audio applications in each VPC. The video traffic contains UDP elephant flows with bandwidth ranging from 2.4Mbps (720P video) to 100Mbps (8K video) [10, 15]. The audio traffic consists of UDP mice flows whose transmission rate ranges from 12.2kbps to 23.85kbps [41].

Figures 9-12 illustrate the performance metrics of Zeta gateways under different burst traffic scenarios. Zeta assigns a gateway cluster to each VPC, while GWZone assigns a master gateway to each host zone. Thus, Zeta can achieve better load balance to deal with various burst traffic. For example, Figure 9 shows that Zeta can reduce the maximum gateway load by 18.5%, 33.9% and 25.2% compared with GWZone in the three applications, respectively. In addition, it is noteworthy that the acknowledgment and retransmission mechanism of MapReduce's TCP flows further increase the gateway load, which leads to the highest gateway load compared with video and audio streams. Moreover, Figure 11 shows the 99th percentile of normalized FCT, which is normalized to the FCT
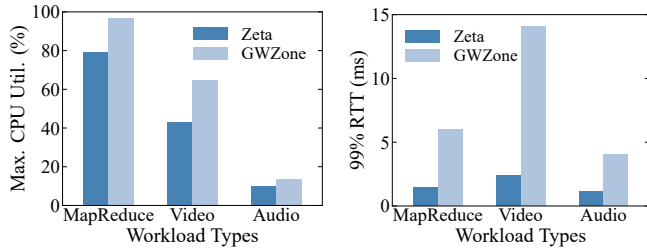
Figure 9: Max. Gateway CPU Utilization vs. Workload Types
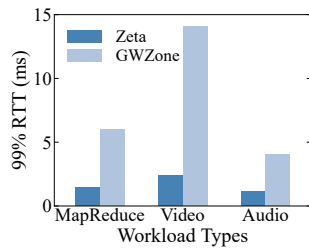
Figure 10: 99% RTT vs. Workload Types
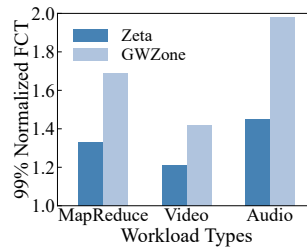
Figure 11: 99% Normalized FCT vs. Workload Types
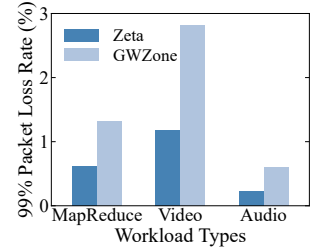
Figure 12: 99% Packet Loss Rate vs. Workload Types
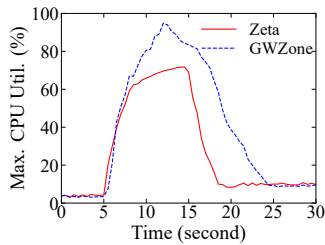
Figure 13: Max. Gateway CPU Utilization over Time
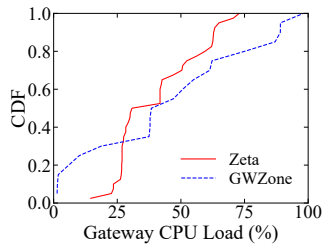
Figure 14: CDF of Gateway CPU Utilization
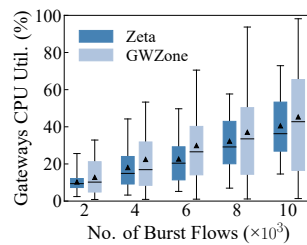
Figure 15: Gateway CPU Utilization vs. No. of Burst Flows
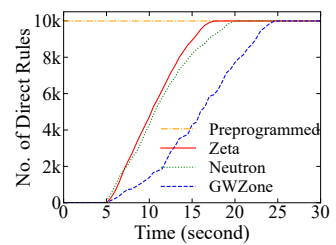
Figure 16: No. of Offloaded Direct Rules over Time

without burst traffic. The 99% normalized FCT achieved by Zeta is 21.3%, 14.8% and 26.8% lower than that of GWZone under three scenarios, respectively. Although the gateway load of audio traffic is low, it mainly consists of mice flows, which will be forwarded by gateways without offloading direct path rules. Thus, the cumulative delay of the audio flows caused by gateway forwarding will be the largest among the three applications, which results in the maximum FCT of audio flows. Besides, we observe from Figure 12 that the 99th percentile of packet loss rate of Zeta under the three scenarios reduces by 53.8%, 58.2% and 63.3% compared with GWZone. The above results prove that Zeta can effectively conquer different burst traffic and avoid gateways overhead.

Furthermore, we evaluate several performance metrics of Zeta in burst video traffic compared with other frameworks, as shown in Figures 13-18. During an interval of 0.5s, we record the CPU utilization of gateways, number of offloaded rules, rule offloading latency and FCT. Specifically, Figure 13 shows the maximum gateway load of Zeta and GWZone in 10k burst video flows. According to the experimental settings, burst traffic are generated randomly in 5-15s, so the gateway load increases sharply at the 5th second. Next, Zeta detects elephant flows faster on the gateways, so it quickly achieves the balance between offloading and newly coming elephant flows. However, the mice flows continue to increase, so the load of Zeta between 7-15s increases slightly on the basis of stability. Meanwhile, the on-host flow detection of GWZone suffers from high latency, and the elephant flows can not be offloaded in time. Thus, the loads of GWZone's gateways increase sharply from 5s to 13s.

To further study how the workload of gateways distributes, we show the gateways' CPU utilization at the 12th second, when Zeta and GWZone both suffer high gateway workload, in Figure 15. Zeta achieves lower average load with more concentrated load distribution than GWZone, which means better load balancing. Figure 14 shows the load CDF of gateways in 10k burst video flows. GWZone's backup gateways are lightly loaded, while 25% master gateways are overloaded (*i.e.*, the CPU load exceeds 80%). The above results show the superiority of Zeta gateway cluster in load balancing.

Figure 16 shows the number of offloaded direct forwarding rules in 10k burst video flows. Due to the latency of the on-host flow detection program, the number of offloaded rules for GWZone increases slowly. The number of Zeta offloading rules is increasing rapidly. Preprogrammed is constant at a high point as its preprogrammed model. The trend of Neutron is similar to Zeta. Figure 18 shows CDF of Normalized FCT in 10k burst video flows. The results are similar to Figure 16. The preprogrammed model performs the best, followed by Zeta and Neutron, while GWZone is the worst.

8.3.2 Fast Recovery from Abnormal Events

We measure the recovery latency of Zeta under abnormal events. Zeta adopts *Multi IPs Migration* for fast recovery, while GWZone updates the default OVS entries on hosts.

Considering that anomaly detection is usually performed by polling, we hope that the delay measurement of failure recovery can avoid the error caused by polling interval. Specifically, we first sequentially send Ping probe every 0.5ms. We
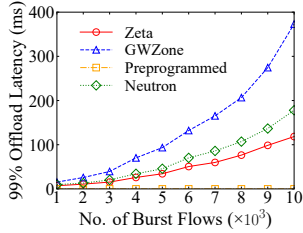
Figure 17: 99% Offload Latency vs. No. of Burst Flows
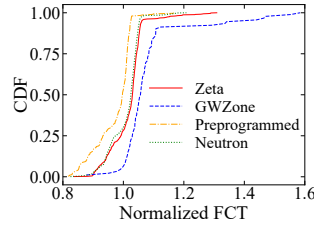
Figure 18: CDF of Normalized FCT
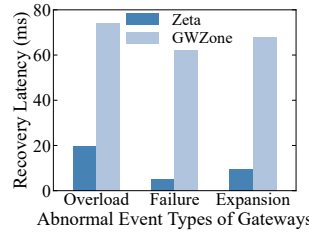
Figure 19: Recovery Latency vs. Abnormal Events
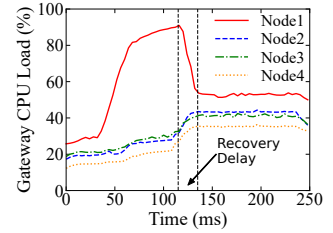
Figure 20: CPU Load of Gateways in a Cluster over Time
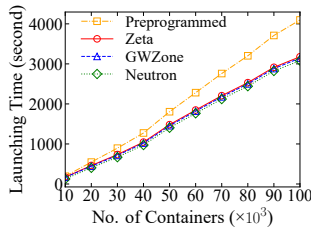
(a) The public cloud
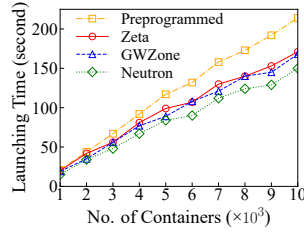
(b) The private cloud

Figure 21: Launching Time vs. No. of Containers

(a) The public cloud

(b) The private cloud

Figure 22: Gateway CPU Utilization vs. No. of Containers

make an artificial abnormal event and notify the controller immediately. Then, the controller performs the IPs Migration. By counting the number of lost packets during the failure recovery, we derive the recovery delay.

From the results in Figure 19, we observe that Zeta can greatly reduce the recovery latency of the three abnormal events compared with GWZone. For example, the gateway failure recovery delay of Zeta is 5.5ms, which is ~10.8× faster than that of GWZone, because GWZone needs to inform each host and update ~100 default entries on each OVS.

Figure 20 illustrates the load status of each gateway in a cluster of Zeta during the overload event. Specifically, the burst flows with default destination of Node1 arrive in 35ms, and the CPU load of Node1 increases rapidly. When the gateway's CPU utilization reaches the 90% threshold, the *Multi IPs Migration* is triggered in 120ms, and three vIPs on Node1 are reassigned to the other three nodes with lighter load. Then, the load of Node1 quickly decreases to a normal level within 19.5ms. It is intuitive that *Multi IPs* can effectively conquer the overload of a single gateway and rapidly adjust the load imbalance of intra-cluster.

8.4 Scalability Evaluation

In this section, we evaluate the scalability of Zeta in both public and private cloud scenarios. We first measure the latency of launching up to 100k container instances. Then, we evaluate the performance metrics of Zeta and GWZone under the large-scale cloud network.

The public cloud scenario contains a large number of instances/VPCs. Based on the transformation (§C.1) of Google

cluster-data [30], we deploy 568 tenants and 1885 VPCs with up to 100k containers on the 20 compute nodes. Each VPC contains 2-364 containers. The private cloud scenarios have a small number of VPCs/tenants, but a VPC may contain a large number of instances. We deploy 52 tenants and 90 VPCs with up to 10k containers on the 20 compute nodes, and each VPC has a number of instances ranging from 2 to 2765.

According to the bandwidth distribution of flows in [22], we let 16% of container pairs communicate, and the traffic intensity of each flow ranges from 10kbps to 1Gbps.

8.4.1 Large-Scale Instances Launching

Figure 21 shows that the on-demand rules offloading model has a lower instance deployment latency compared with the preprogrammed model when spawning a large number of instances in a large-scale cloud network. For example, when launching 100k containers in the public cloud environment, Zeta spends 3178 seconds and installs 12k default forwarding rules, while Preprogrammed spends 4097 seconds and programs a total of 3.4M rules. That is, Zeta reduces the launching time by 24% and the number of rules by 278× compared with Preprogrammed. The reason for the above results is that the on-demand rules offloading can avoid pre-installing numerous entries for instances that never communicate with each other, thus it reduces the latency of instances launching.

8.4.2 Large-Scale Instances Communication

Figures 22-24 show the advantages of Zeta gateway cluster under large-scale networks. As shown in Figure 22, the average

(a) The public cloud (b) The private cloud
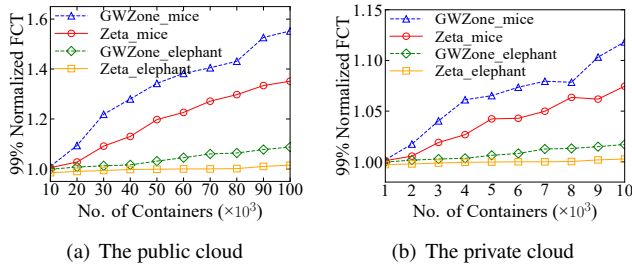
Figure 23: 99% Normalized FCT vs. No. of Containers

(a) The public cloud (b) The private cloud

Figure 24: Packet Loss Rate vs. No. of Containers

load of Zeta gateways is close to that of GWZone. However, Zeta gateways achieve more concentrated load distribution than GWZone and there is a big gap between maximum and minimum load of GWZone gateways, which means the superiority of Zeta gateway cluster in load balancing.

Next, we evaluate the impact of Zeta and GWZone gateways on FCT. The Normalized FCT of elephant flows and mice flows are calculated respectively. Figure 23 shows that though Zeta and GWZone have the similar normalized FCT, Zeta still outperforms GWZone by 7% in public cloud scenario, as there is no flow detection load on hosts. In addition, the FCT of elephant flows are both smaller than that of mice flows, because the elephant flows will be forwarded directly.

Finally, we evaluate the packet loss rate of Zeta and GW-Zone with offloaded elephant flows and non offloaded mice flows to prove the scalability of Zeta. Figure 24 shows that the packet loss rate of Zeta is lower than that of GWZone because of the better load balancing effect of Zeta gateway cluster. For example, in public cloud with the network scale of 100k containers, the packet loss rate of elephant flows and mice flows of Zeta is 24% and 37% lower than that of GWZone, respectively. In addition, the packet loss rate of elephant flows is higher than that of mice flows. The reason is that these elephant flows will be forwarded by the gateways at the beginning, and burst traffic will cause the gateways overload, resulting in a higher packet loss rate. Therefore, the packet loss of elephant flows is mainly concentrated in the initial gateway forwarding period, and the packet loss of direct path forwarding after offloading will be significantly reduced.

9 Related Work

Cloud and datacenter virtual networks. There are a multitude of researches on the cloud/datacenter virtual networks, including control plane [21, 26, 35, 40, 73] and data plane [22, 39, 55, 61]. As a crucial solution, overlay network adopts tunnel encapsulation protocols (*e.g.*, VXLAN [48], NVGRE [70], Geneve [33], etc) to build the scalable and flexible virtual networks. Virtual network devices (*e.g.*, vSwitch [58, 76], vRouter [69] and gateway [22, 57]) are essential in the cloud networks, as they are dedicated to provide efficient, secure and stable connections for tenants in clouds. In this paper, we
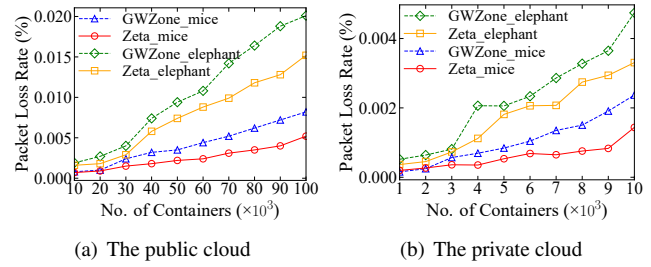
focus on improving the robustness of east-west forwarding with the designs of gateway cluster and multi IPs migration.

High performance and programmable data plane. Data plane is the most performance-critical part of the cloud networks, which is usually accelerated with specialized hardware components and sophisticated software methods [9]. In hardware, ASIC [57, 75], FPGA [16, 28, 46, 61] and network processor [51, 53] can provide high-throughput and low-latency packet processing. In contrast, software methods have the advantage of fast and flexible iteration, including DPDK [24], XDP [36], Netmap [62], etc. Though XDP is not the first mover in this area, we choose XDP as the data plane of Zeta, because of its integration with Linux kernel, interaction with other kernel components and similar speed as DPDK.

eBPF and its applications. eBPF is an instruction set and an execution environment inside the Linux kernel [79]. It enables injecting custom code into the kernel through the hooks. eBPF is extensively used in security [25], tracing [11] and networking [20]. XDP is one of the most widely used eBPF hooks for high-performance packet processing that can bypass the kernel network stack [36].

10 Conclusion and Future Work

In this paper, we propose a scalable and robust east-west communication framework in large-scale clouds, called Zeta. Comprehensive experiment results show high robustness and scalability of Zeta. For example, Zeta speeds up the gateway failure recovery by 10.8× compared with the existing solutions. In future, we will optimize the timeout mechanism of eBPF map to reduce the impact on forwarding performance.

Acknowledgments

References

[1] Calico Project, 2022. https://www.tigera.io/project-calico/.

[2] eBPF Maps, 2022. https://ebpf.io/what-is-ebpf/#maps.

[3] Etcd: A distributed, reliable key-value store, 2022. https://etcd.io/.

[4] Flannel Project, 2022. https://github.com/flannel-io/flannel.

[5] David Ahern. XDP and the cloud: Using XDP on hosts and VMs, 2020. https://legacy.netdevconf.info/0x14/pub/slides/24/netdev-0x14-XDP-and-the-cloud.pdf.

[6] Amazon AWS. AWS Nitro System, 2022. https://aws.amazon.com/ec2/nitro/.

[7] Victor Bahl. Emergence of micro datacenter (cloudlets/edges) for mobile computing. *Microsoft Devices & Networking Summit 2015*, 5, 2015.

[8] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280, 2010.

[9] Roberto Bifulco and Gábor Rétvári. A survey on the programmable data plane: Abstractions, architectures, and open problems. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, pages 1–7. IEEE, 2018.

[10] Kashif Bilal and Aiman Erbad. Impact of multiple video representations in live streaming: A cost, bandwidth, and QoE analysis. In *2017 IEEE International Conference on Cloud Engineering (IC2E)*, pages 88–94. IEEE, 2017.

[11] bpftrace. High-level tracing language for Linux systems, 2022. https://bpftrace.org/.

[12] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. hXDP: Efficient Software Packet Processing on FPGA NICs. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 973–990, 2020.

[13] Tuan Anh Bui and Marco Canini. *Cloud network performance analysis: an OpenStack case study*. PhD thesis, Master's thesis, Université Catholique de Louvain, 2016.

[14] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. Understanding host network stack overheads. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 65–77, 2021.

[15] Christopher Canel, Thomas Kim, Giulio Zhou, Conglong Li, Hyeontaek Lim, David G Andersen, Michael Kaminsky, and Subramanya R Dulloor. Scaling video analytics on constrained edge nodes. *arXiv preprint arXiv:1905.13536*, 2019.

[16] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[17] Qixiang Cheng, Meisam Bahadori, Madeleine Glick, Sébastien Rumley, and Keren Bergman. Recent advances in optical technologies for data centers: a review. *Optica*, 5(11):1354–1370, 2018.

[18] Sean Choi, Boris Burkov, Alex Eckert, Tian Fang, Saman Kazemkhani, Rob Sherwood, Ying Zhang, and Hongyi Zeng. Fboss: building switch software at scale. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 342–356, 2018.

[19] Cilium. BPF and XDP Reference Guide, 2022. https://docs.cilium.io/en/latest/bpf/.

[20] Cilium. eBPF-based Networking, Security, and Observability, 2022. https://cilium.io/.

[21] Andrew R Curtis, Jeffrey C Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. Devoflow: Scaling flow management for high-performance networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, pages 254–265, 2011.

[22] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, et al. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 373–387, 2018.

[23] Yaozu Dong, Xiaowei Yang, Jianhui Li, Guangdeng Liao, Kun Tian, and Haibing Guan. High performance network virtualization with SR-IOV. *Journal of Parallel and Distributed Computing*, 72(11):1471–1480, 2012.

[24] DPDK. Data Plane Development Kit, 2022. https://www.dpdk.org/.

[25] Falco. Cloud Native Runtime Security, 2022. https://falco.org/.

[26] Andrew D Ferguson, Steve Gribble, Chi-Yao Hong, Charles Edwin Killian, Waqar Mohsin, Henrik Muehe, Joon Ong, Leon Poutievski, Arjun Singh, Lorenzo Vicisano, et al. Orion: Google's software-defined networking control plane. In *NSDI*, pages 83–98, 2021.

[27] Ross Finlayson, Timothy Mann, JC Mogul, and Marvin Theimer. RFC0903: Reverse Address Resolution Protocol, 1984.

[28] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure accelerated networking: Smartnics in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, 2018.

[29] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing. In *NSDI*, pages 487–501, 2021.

[30] Google. Google cluster-data, 2020. https://github.com/google/cluster-data.

[31] Google Cloud. Containers at Google, 2022. https://cloud.google.com/containers.

[32] Clinton Gormley and Zachary Tong. *Elasticsearch: the definitive guide: a distributed real-time search and analytics engine*. " O'Reilly Media, Inc.", 2015.

[33] Jesse Gross, T Sridhar, P Garg, C Wright, I Ganga, P Agarwal, K Duda, D Dutt, and J Hudson. Geneve: Generic network virtualization encapsulation. *IETF draft*, 2014.

[34] Gurobi. The Fastest Solver, 2022. https://www.gurobi.com/.

[35] Soheil Hassas Yeganeh and Yashar Ganjali. Kandoo: a framework for efficient and scalable offloading of control applications. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 19–24, 2012.

[36] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The express data path: Fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th international conference on emerging networking experiments and technologies*, pages 54–66, 2018.

[37] iPerf. The TCP, UDP and SCTP network bandwidth measurement tool, 2022. https://iperf.fr/.

[38] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. The nature of data center traffic: measurements & analysis. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement*, pages 202–208, 2009.

[39] Teemu Koponen, Keith Amidon, Peter Balland, Martín Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Paul Ingram, Ethan Jackson, et al. Network virtualization in multi-tenant datacenters. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 203–216, 2014.

[40] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, et al. Onix: A distributed control platform for large-scale production networks. In *OSDI*, volume 10, pages 1–6, 2010.

[41] Linda Kozma-Spytek, Paula Tucker, and Christian Vogler. Voice telephony for individuals with hearing loss: The effects of audio bandwidth, bit rate and packet loss. In *The 21st International ACM SIGACCESS Conference on Computers and Accessibility*, pages 3–15, 2019.

[42] kubernetes Project. kubernetes Eviction Policy, 2022. https://kubernetes.io/docs/concepts/scheduling-eviction/eviction-policy/.

[43] kubernetes Project. Kubernetes Scheduler, 2022. https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/.

[44] Martin KaFai Lau. bpf: Improve LRU map lookup performance, 2017. https://patchwork.ozlabs.org/project/netdev/cover/20170901062713.1842249-1-kafai@fb.com/.

[45] Jeongkeun Lee, Yoshio Turner, Myungjin Lee, Lucian Popa, Sujata Banerjee, Joon-Myung Kang, and Puneet Sharma. Application-driven bandwidth guarantees in datacenters. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 467–478, 2014.

[46] Bojie Li, Kun Tan, Layong Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 1–14, 2016.

[47] Yilong Li, Seo Jin Park, and John K Ousterhout. Millisort and milliquery: Large-scale data-intensive computing in milliseconds. In *NSDI*, pages 593–611, 2021.

[48] Mallik Mahalingam, Dinesh G Dutt, Kenneth Duda, Puneet Agarwal, Lawrence Kreeger, T Sridhar, Mike Bursell, and Chris Wright. Virtual extensible local area network (vxlan): A framework for overlaying virtualized layer 2 networks over layer 3 networks. *RFC*, 7348:1–22, 2014.

[49] Ilias Marinos, Robert NM Watson, and Mark Handley. Network stack specialization for performance. *ACM SIGCOMM Computer Communication Review*, 44(4):175–186, 2014.

[50] Jaehyun Nam, Seungsoo Lee, Hyunmin Seo, Phil Porras, Vinod Yegneswaran, and Seungwon Shin. Bastion: A security enforcement network stack for container networks. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 81–95, 2020.

[51] Netronome. Agilio CX SmartNICs, 2022. https://www.netronome.com/products/agilio-cx/.

[52] B Niven-Jenkins, D Brungard, M Betts, N Sprecher, and S Ueno. Requirements of an MPLS transport profile, 2009.

[53] Nvidia/Mellanox. Nvidia BlueField Data Processing Units, 2022. https://www.nvidia.com/en-us/networking/products/data-processing-unit/.

[54] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. Stateless datacenter load-balancing with beamer. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 125–139, 2018.

[55] OpenStack Project. OpenStack Basic Networking, 2022. https://docs.openstack.org/neutron/latest/admin/intro-basic-networking.html.

[56] Marcus Oppitz and Peter Tomsu. Software defined virtual networks. In *Inventing the Cloud Century*, pages 149–200. Springer, 2018.

[57] Tian Pan, Nianbing Yu, Chenhao Jia, Jianwen Pi, Liang Xu, Yisong Qiao, Zhiguo Li, Kun Liu, Jie Lu, Jianyuan Lu, et al. Sailfish: accelerating cloud-scale multi-tenant multi-service gateways with programmable switches. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 194–206, 2021.

[58] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The design and implementation of open vswitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 117–130, 2015.

[59] Rahul Potharaju and Navendu Jain. Demystifying the dark side of the middle: A field study of middlebox failures in datacenters. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 9–22, 2013.

[60] Konstantinos Poularakis, Qiaofeng Qin, Liang Ma, Sastry Kompella, Kin K Leung, and Leandros Tassiulas. Learning the optimal synchronization rates in distributed SDN control architectures. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 1099–1107. IEEE, 2019.

[61] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 13–24. IEEE, 2014.

[62] Luigi Rizzo. Netmap: A novel framework for fast packet I/O. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 101–112, 2012.

[63] Tiago Rosado and Jorge Bernardino. An overview of openstack architecture. In *Proceedings of the 18th International Database Engineering & Applications Symposium*, pages 366–367, 2014.

[64] Arsalan Saghir and Tahir Masood. Performance evaluation of openstack networking technologies. In *2019 International Conference on Engineering and Emerging Technologies (ICEET)*, pages 1–6. IEEE, 2019.

[65] Ken-ichi Sato, Hiroshi Hasegawa, Tomonobu Niwa, and Toshio Watanabe. A large-scale wavelength routing optical switch for data center networks. *IEEE Communications Magazine*, 51(9):46–52, 2013.

[66] Keith Seymour, Hidemoto Nakada, Satoshi Matsuoka, Jack Dongarra, Craig Lee, and Henri Casanova. Overview of GridRPC: A remote procedure call API for grid computing. In *International Workshop on Grid Computing*, pages 274–278. Springer, 2002.

[67] Leah Shalev, Hani Ayoub, Nafea Bshara, and Erez Sabbag. A cloud-optimized transport protocol for elastic and scalable hpc. *IEEE Micro*, 40(6):67–73, 2020.

[68] Danfeng Shan, Fengyuan Ren, Peng Cheng, Ran Shu, and Chuanxiong Guo. Observing and mitigating microburst traffic in data center networks. *IEEE/ACM Transactions on Networking*, 28(1):98–111, 2019.

[69] Hua Shao, Xiaoliang Wang, Yuanwei Lu, Yanbo Yu, Shengli Zheng, and Youjian Zhao. Accessing cloud with disaggregated software-defined router. In *NSDI*, pages 1–14, 2021.

[70] Murari Sridharan. Nvgre: Network virtualization using generic routing encapsulation. *draft-sridharan-virtualization-nvgre-00. txt*, 2011.

[71] Piyush Raman Srivastava and Saket Saurav. Networking agent for overlay L2 routing and overlay to underlay external networks L3 routing using OpenFlow and Open vSwitch. In *2015 17th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, pages 291–296. IEEE, 2015.

[72] Dimitri Staessens, Sachin Sharma, Didier Colle, Mario Pickavet, and Piet Demeester. Software defined networking: Meeting carrier grade requirements. In *2011 18th IEEE workshop on local & metropolitan area networks (LANMAN)*, pages 1–6. IEEE, 2011.

[73] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky HY Wong, and Hongyi Zeng. Robotron: Top-down network management at facebook scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 426–439, 2016.

[74] The kernel development community. Linux TUN/TAP Device, 2022. https://www.kernel.org/doc/html/latest/networking/tuntap.html.

[75] Barefoot Tofino. World's fastest P4-programmable Ethernet switch ASICs, 2018.

[76] William Tu, Yi-Hung Wei, Gianni Antichi, and Ben Pfaff. Revisiting the Open vSwitch dataplane ten years later. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 245–257, 2021.

[77] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. A placement vulnerability study in multi-tenant public clouds. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 913–928, 2015.

[78] Luis Velasco. Recovery mechanisms in ASON/GMPLS networks. *Universitat Politècnica de Catalunya (UPC), Barcelona, Spain*, 2009.

[79] Marcos AM Vieira, Matheus S Castanho, Racyus DG Pacífico, Elerson RS Santos, Eduardo PM Câmara Júnior, and Luiz FM Vieira. Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. *ACM Computing Surveys (CSUR)*, 53(1):1–36, 2020.

[80] Jason Wang. Accelerating VM networking through XDP, 2017. https://events19.linuxfoundation.cn/wp-content/uploads/2017/11/Accelerating-VM-Networking-through-XDP_Jason-Wang.pdf.

[81] Jingzhou Wang, Gongming Zhao, Hongli Xu, Yutong Zhai, Qianyu Zhang, He Huang, and Yongqiang Yang. A robust service mapping scheme for multi-tenant clouds. *IEEE/ACM Transactions on Networking*, 2021.

[82] Weina Wang, Kai Zhu, Lei Ying, Jian Tan, and Li Zhang. Maptask scheduling in mapreduce with data locality: Throughput and heavy-traffic optimality. *IEEE/ACM Transactions On Networking*, 24(1):190–203, 2014.

[83] Lizhao You, Hao Tang, Jiahua Zhang, and Xiao Li. Fast configuration change impact analysis for network overlay data center networks. In *4th Asia-Pacific Workshop on Networking*, pages 8–15, 2020.

[84] Gongming Zhao, Hongli Xu, Shigang Chen, Liusheng Huang, and Pengzhan Wang. Joint optimization of flow table and group table for default paths in sdns. *IEEE/ACM Transactions on Networking*, 26(4):1837–1850, 2018.

A Additional Details of Cluster Mapping

A.1 Empirical Formula for Tenant Constraint

We use $C = \{c_1, c_2, ..., c_n\}$ to denote the gateway clusters, where $n = |C|$ is the number of clusters. In addition, let I_t denote the number of instance owned by tenant $t \in T$. Then, we use the following empirical formula to set the tenant constraint k:

$$k = \lceil \frac{max_{t \in T}\{I_t\}}{\sum_{t \in T} I_t} \times n \rceil + 1 \tag{2}$$

For example, when our testbed §8.4 contains 100k container instances, the largest tenant has 27652 instances. We set the tenant constraint $k = 4$, which means the VPCs of a tenant will be mapped to at most 4 gateway clusters.

A.2 Rounding-Based Algorithm

To solve the problem in Eq. (1), we propose a rounding-based gateway cluster mapping (RGCM) algorithm for the GCM problem. The RGCM algorithm includes two steps. The first step is to construct a relaxed version of GCM, named LP-GCM, by relaxing the variable binary constraints. Specifically, LP-GCM assumes that each flow can be splitable and forwarded to multiple gateway clusters. Since LP-GCM is a linear programming, we can derive the fractional solutions $\{\tilde{x}_v^c\}$ and $\{\tilde{y}_t^c\}$ with an optimization solver, such as Gurobi [34]. The optimal fractional result is denoted as $\tilde{\lambda}$.

The second step is to derive the integer solutions with rounding scheme. The integer solutions are denoted as $\{\hat{x}_v^c\}$
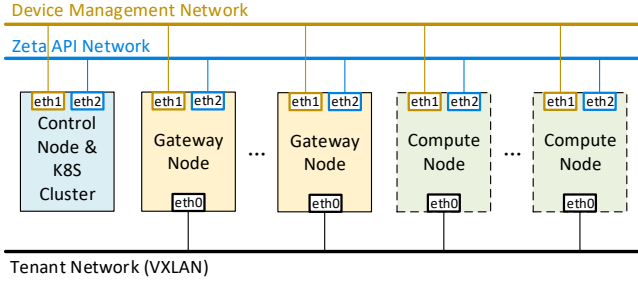
Figure 25: Best Practice for Zeta Physical Deployment.

Figure 26: Early Version of KVM-based Gateway.

and $\{\widehat{y}_t^c\}$. For each tenant $t \in T$, RGCM first sorts each gateway cluster $c \in C$ by the value of $\widetilde{y}_t^c$ in the descending order. Then RGCM sets the top k maximum $\widehat{y}_t^c$ to 1, which means that the traffic of tenant t can be processed by these k gateway clusters. The set of clusters that are available to the tenant t is denoted as C_t, *i.e.*, $C_t = \{c | \widehat{y}_t^c = 1, c \in C\}$, where $|C_t| = k$. When variables $\{\widehat{y}_t^c\}$ have been determined, RGCM will assign a gateway cluster to each VPC $v \in V$, *i.e.*, determine variables $\{\widehat{x}_v^c\}$. For each VPC $v \in V$, the algorithm selects a cluster $c \in C_t$ with the least burden and sets variable $\widehat{x}_v^c$ to 1.

While solving a linear programming might take a long time for a large network, we note that tenants/VPCs/instances are deployed incrementally, and the number of VPCs/tenants is usually much smaller than that of instances. For example, if hundreds of thousands of instances boot up at the same time, the corresponding VPCs are thousands and the corresponding tenants are hundreds. We utilize Gurobi solver [34] to run the RGCM algorithm on a server equipped with a 10-core Intel i9-10900 CPU. The solution time is 1.15s for the network with 10 gateway clusters, 568 tenants and 1885 VPCs in §8.4, which is acceptable compared to the VPC/instance deployment time.

B Additional Implementation Details

B.1 eBPF Map Size

In the current Linux kernel implementation, the memory usage of an eBPF hash map grows with its entry number. However, the maximum entry size is bounded by the `max_entries` defined by XDP/eBPF program during map initialization. The user space function `bpf_map__resize()` can resize an eBPF map only before it is initialized in the kernel. Unfortunately, we cannot resize an eBPF map after it is created. We have to deploy a new XDP/eBPF program to reinitialize the map size.

Thus, the number of instances that a gateway cluster can serve is limited by the `max_entries` of the eBPF maps. For example, the `endpoint` hash map in Zeta stores instance forwarding rules and its `max_entries` is set to 128*1024 ($\sim$131k). To avoid the above limitation, we can set a larger entry number for the `endpoint` hash map, such as 1024*1024 ($\sim$1M). In addition, the key size and value size of one
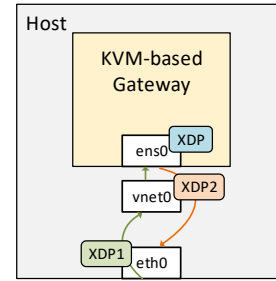
endpoint entry is 8 bytes and 16 bytes, respectively. The total memory size of 1M entries is only 24MB.

B.2 Best Practice for Physical Deployment

Zeta is usually deployed as two self-contained parts: (i) One Kubernetes micro-service hosting Cluster Controller services; (ii) One Gateway Cluster for Zeta data plane, which is based on physical machines in production environment.

Figure 25 illustrates the best practice of Zeta deployment, which includes a control node, several gateway nodes and compute nodes. The leftmost control node deploys the management service of the cloud platform and Kubernetes cluster hosting Zeta Controller. The middle ones are gateway nodes, each of which deploys DFT and FWD modules. The `eth1` interfaces of all nodes access the Device Management Network. In addition, we use separate interfaces for the Zeta API Network and Tenant Network, which prevents massive tenants' traffic from blocking the control messages. The Zeta API Network is responsible for sending the operation instructions and reporting status information, including OAM packets, IPs allocation/migration policies and gateways' load information. The Tenant Network transmits the east-west traffic through the VXLAN tunnel [48] for tenant instances.

B.3 Additional Details of Virtual Deployment

In the early development of Zeta, we use TUN/TAP device [74] as the NICs of KVM-based gateways. In addition to deploying XDP in the KVM-based gateways, we also deploy additional XDP programs on the physical machines to accelerate the host-VM datapath [5, 80]. As shown in Figure 26, we attach `XDP1` to the NIC (*i.e.*, `eth0`) of the physical machine to accelerate the host-VM ingress traffic. We attach `XDP2` to the TAP device (*i.e.*, `vnet0`) on the physical machine to accelerate the VM-host egress traffic.

However, Zeta suffers from the poor forwarding performance. For example, the packet forwarding rate of a KVM-based gateway equipped with 4 vCPUs is only 1.36Mpps. The reason is that attaching XDP program to VM's NIC will affect the function of TAP device in host and lead to a significant hit on VM RX performance [5].

Finally, Zeta adopts SR-IOV [23] for KVM-based gateways. Although the driver of Intel XL710 VF (i.e., iavf) does not support XDP Native mode, and Zeta adopts XDP Generic mode with reduced performance in KVM-based gateways [19, 36]. We obtain an acceptable forwarding performance. For example, the pure forwarding rate of one virtual core is 0.86Mpps under 2k entries, which drops 54% compared with one physical core with XDP Native mode.

C Additional Evaluation Details

C.1 Transformation of Google cluster-data

We query the `a.CollectionEvents` table of Google cluster trace and obtain the mapping of `<user,machine,job>` [30]. The machine number is 10001 and the user number is 1952. Considering that we only have 20 compute nodes, while there are 10001 machines in the table. Thus, we merge the jobs of every 500 machines to one compute nodes.

C.2 Linear Scaling Throughput of Gateways
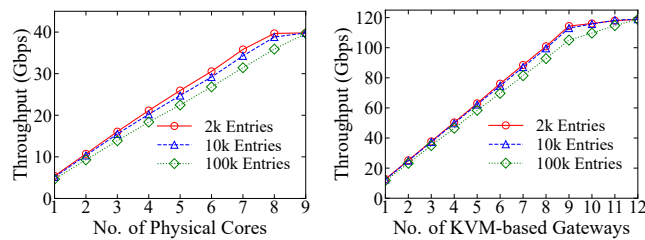
Figure 27: Throughput vs. No. of Physical Cores

Figure 28: Throughput vs. No. of KVM-based Gateways

Linear Scaling Throughput. Figures 27-28 show that the total throughput will scale linearly with the increasing number of physical cores and KVM-based gateways. Specifically, when the inner packet size is 512 bytes and the number of entries in eBPF maps is 2k, the throughput of a physical core is 5.4Gbps, and 8 physical cores will hit the NIC's bandwidth limit of the physical machine at 40Gbps. The throughput of a KVM-based gateway with 4 vCPU is 12.7Gbps, and 9 KVM-based gateways will nearly reach the NICs' total bandwidth limit of the 3 physical gateway machines at 120Gbps. In addition, the timeout mechanism of maps for flow statistics leads to throughput degradation with the number of entries increases. We will try to optimize this issue in future work. The linear scaling throughput of Zeta gateways greatly enhances the scalability of Zeta gateway clusters.

Aquila: A unified, low-latency fabric for datacenter networks

Dan Gibson, Hema Hariharan, Eric Lance, Moray McLaren, Behnam Montazeri, Arjun Singh,
Stephen Wang, Hassan M. G. Wassel, Zhehua Wu, Sunghwan Yoo, Raghuraman Balasubramanian,
Prashant Chandra, Michael Cutforth, Peter Cuy, David Decotigny, Rakesh Gautam,
Alex Iriza, Milo M. K. Martin, Rick Roy, Zuowei Shen, Ming Tan, Ye Tang, Monica Wong-Chan,
Joe Zbiciak, Amin Vahdat
Google Inc.
aquila-nsdi2022@google.com

Abstract

Datacenter workloads have evolved from the data intensive, loosely-coupled workloads of the past decade to more tightly coupled ones, wherein ultra-low latency communication is essential for resource disaggregation over the network and to enable emerging programming models.

We introduce *Aquila*, an experimental datacenter network fabric built with ultra-low latency support as a first-class design goal, while also supporting traditional datacenter traffic. Aquila uses a new Layer 2 cell-based protocol, GNet, an integrated switch, and a custom ASIC with low-latency Remote Memory Access (RMA) capabilities co-designed with GNet. We demonstrate that Aquila is able to achieve under 40 μs tail fabric Round Trip Time (RTT) for IP traffic and sub-10 μs RMA execution time across hundreds of host machines, even in the presence of background throughput-oriented IP traffic. This translates to more than 5x reduction in tail latency for a production quality key-value store running on a prototype Aquila network.

1 INTRODUCTION

There has been tremendous progress in datacenter networking over the past decade, with fundamental advances in the control plane [18,27,44,49], the rise of commodity silicon arranged in non-blocking topologies [4,23,36,49], network management and verification [7,8,29,41], and highly available network design techniques [21]. Taken together, the community is now in a place where cost-effective, easy-to-manage, and scalable network designs and deployments are becoming common in industry. Plentiful network bandwidth at the scale of clusters of tens of thousands servers [49] can be leveraged for large-scale hyperscalers and the services they host.

However, all of these advances come while assuming TCP-based congestion control and Ethernet Layer 2 protocols. This Layer 2-4 stack has been incredibly robust and resilient through many decades of deployment and incremental evolution. However, we are seeing a new impasse in the datacenter [12] where advances in distributed computing are increasingly limited by the lack of performance predictability and isolation in multi-tenant datacenter networks. Two to three orders of magnitude performance difference [15] in what network fabric designers aim for and what applications can expect and program to is not uncommon, severely limiting the pace of innovation in higher-level cluster-based distributed systems.

Such concerns are amplified when considering the state of supercomputing/HPC clusters [17] and emerging machine learning pods [22,28] where individual applications benefit from low-latency RDMA [16,51], collective operations [46], and tightly integrated compute and communication capabilities. The key differences in these more specialized settings relative to production datacenter environments include: i) the ability to assume single tenant deployments or at least space sharing rather than time sharing; ii) reduced concerns around failure handling; and iii) a willingness to take on backward incompatible network technologies including wire formats.

Recent research efforts into disaggregated rack-scale architectures [13,34] further highlight some of these challenges: can the same NICs and switches supporting host-to-host communication across the wide area support, for example, SSD and GPU devices at a much smaller radius? Is the disaggregation network necessarily a separate dedicated fabric or can it be multiplexed with TCP/IP traffic destined to remote hosts potentially 100ms or more away? While there is some appeal to running a second (or third) network dedicated for an individual use case, the control and, as importantly, the management overhead of each network introduces a cyclic dependency where the second network is not worthwhile relative to the status quo until the underlying technology is proven/mature. However, there is no opportunity to iterate on the alternate technology because doing so is cost and complexity negative for a number of generations into the future because applications would have to evolve substantially before demonstrating end-to-end wins on the new hardware.

The need for backward compatibility combined with challenges in deploying niche "bag on the side" networks threatens a new ossification in datacenter networking and dis-

tributed systems where we are left with programming to the lowest common denominator of TCP transports and commodity Ethernet switches with associated latency, CPU efficiency, and isolation limitations.

In this paper, we present a first exploration of an alternative tightly-coupled (or *Clique*-based) datacenter architecture, *Aquila*, a hardware implementation supporting predictable, high-bandwidth, and ultra-low latency communication. In our approach, datacenter networks consist of dozens of Cliques, each hosting approximately 1-2k network ports. Cliques interoperate with one another at the datacenter interface (e.g., the spine layer of existing Clos-based datacenter networks) through standard Ethernet and IP. However, within a Clique, any transport and Layer 2 network protocol may be deployed. Applications that fit within the boundaries of an individual Clique can assume Clique-local capability, including robust RDMA, predictable low-latency communication, device disaggregation, support for ML aggregation primitives, etc. We assume IP-based transport for communication between Cliques, which means that any intra-Clique communication primitives and innovations must live alongside standard transports. Cliques then become the unit of deployment, innovation, and homogeneity, allowing for incremental, backward-compatible deployment into existing datacenters. A Clique is also sufficiently large to host all but the largest of individual distributed systems, especially as we move to hundreds of compute cores per server.

Aquila, our first Clique implementation based around a custom in-house ASIC and communication software, consists of a cell-switched non-Ethernet substrate, GNet.

- Aquila networks are built from individual silicon components that serve as both NIC and a portion of the traditional Top of Rack (ToR) switch; each *ToR-in-NIC* (TiN) chip attaches to hosts and directly to other TiN chips to realize a cost-effective network built from a single, replicated silicon component, rather than distinct NIC and switch silicon components from separate vendors.
- GNet provides the illusion of Ethernet to hosts within Aquila, as well as to non-Aquila networking components outside the scope of the Aquila Clique, by terminating Ethernet at the Aquila network boundary and tunneling traffic across a fully-custom, self-defending, near-lossless L2 substrate.
- Aquila further reduces cost by realizing a direct-network rather than an indirect (Clos) topology. To fully unlock the capabilities of its Dragonfly topology [30], and freed from the de facto constraints imposed by Ethernet, Aquila leverages adaptive routing to deliver full point-to-point bandwidth between host-pairs by leveraging multiple non-minimal paths.
- Aquila delivers data in small chunks called *cells*, rather than packets, thereby optimizing for latency of small exchanges like those used by distributed systems built on RDMA and similar technologies [51]. Its extremely tight

integration between NIC and network allows for ultra-low RMA-read capability (4us median) between the memory systems of up to 1152 hosts.

Aquila's design departs from traditional Ethernet fabrics in several ways: i) links use credit-based flow-control; ii) switch buffering is shallow; and iii) solicitation bounds end-to-end admission. Any one of these tenets in Ethernet would be problematic, but taken together, they form a cohesive design. For instance, flow-controlled near-lossless links can give rise to tree saturation, especially with shallow buffering, but end-to-end admission control bounds the size and spread of such trees, and ensures they are transient. Similarly, admission control breaks down when drops are likely, but link-level flow control makes drops very rare, and in turn enables the use of shallow buffering in switching elements, since overrun is not possible.

We present the detailed design, implementation, and evaluation of Aquila. Aquila is not the final word in Clique design; in fact, our first experience with the Aquila system suggests a number of areas for improvement in future generations. We hope, however, that the approach of bringing vertical integration including the host software stack, the NIC, and the switch along with a Clique-based datacenter architecture will enable new models of datacenter innovation along with new capabilities to distributed systems that can assume cutting edge rather than lowest common denominator communication and disaggregation capability within the boundary of thousands of servers and hundreds of thousands of cores.

2 OBJECTIVES AND OUR APPROACH

Aquila's design departures from Ethernet are grounded in a set of common objectives, described below. Taken individually, these design choices–e.g., flow control, custom Layer 2–would be hard to apply to an existing network incrementally. But in concert, Aquila's features realize a complete, performant design point.

Sustainable hardware development. To sustain the hardware development effort with a modest sized team, we chose to build a single chip with both NIC and switch functionality in the same silicon. Our fundamental insight and starting point was that a *medium-radix* switch could be incorporated into existing NIC silicon at modest additional cost and that a number of these resulting NIC/switch combinations called *ToR-in-NIC* (*TiN*) chips could be wired together via a copper backplane in a *pod*, an enclosure the size of a traditional Top of Rack (ToR) switch. Servers could then connect to the pod via PCIe for their NIC functionality. The TiN switch would provide connectivity to other servers in the same Clique via an optimized Layer 2 protocol, GNet, and to other servers in other Cliques via standard Ethernet. The inset in Figure 1 summarizes the major components of TiN.

Cost effective, non-blocking topology. For efficiency and low latency, we selected a direct connect topology, Dragonfly, a well-studied topology that minimizes the number of
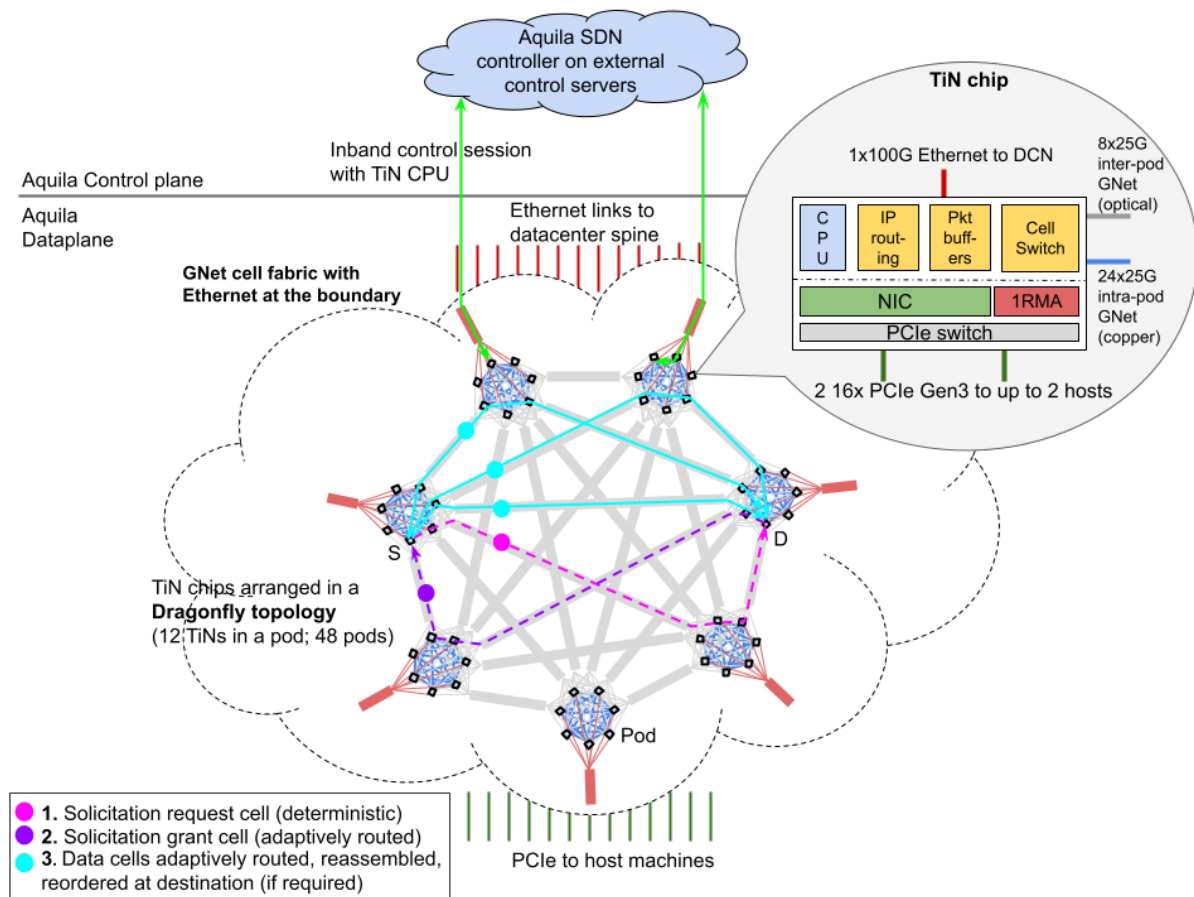
Figure 1: Aquila Clique dataplane and control plane overview. The ToR-in-NIC (TiN) chip is expanded in the top right inset. TiN chips are arranged in a cell-switched Dragonfly network, connecting via Ethernet to the rest of the datacenter network's (DCN) spine switches and to host machines via PCIe. Conventional Ethernet/IP packets are split into cells at ingress after a round of solicitation per packet and reassembled and re-ordered at the fabric's egress. The co-designed 1RMA protocol injects cells directly into the cell network, extending memory accesses across the Clique. The Aquila SDN controller configures and manages TiN switches inband, via the DCN.

long optical links in the network while still providing non-blocking bandwidth for uniform random traffic patterns, with 2:1 over-subscription for worst-case adversarial traffic. Figure 1 illustrates a simplified Dragonfly topology where TiNs within a pod are fully connected in a mesh, and multiple pods are likewise connected all-to-all to form a tightly-coupled Clique network. The largest Aquila network supports 12 TiNs in a pod with 48 pods, serving up to 1152 host machines.

Combining NIC and ToR into the single TiN chip was a less costly path to innovation than separate NIC and switch ASIC programs, and a design realized from a common single component was intended to streamline inventory management for Aquila. Further, we implemented an optional capability to allow pairs of host machines to share a single TiN, halving the normalized cost of ownership for networking per host, trading off reduced sustained bandwidth provisioning per machine.

Ultra low-latency network. To optimize for ultra-low latency, under load and in the tail, Aquila implements cell-based communication with shallow buffering for cells within the network, flow controlled links for near lossless cell delivery, and hardware adaptive routing to react in nanoseconds to link

failures and to keep the network load balanced even at high loads. To ensure recipients are not overwhelmed, Aquila implements end-to-end solicitation for each packet at ingress, which guarantees that resources are available at the destination TiN before the packet can be split into cells and transmitted from the source TiN. We built these latency-guarding features into Aquila's Layer 2 protocol, GNet. As depicted in Figure 1, while the Aquila network fabric presents an Ethernet packet interface at its boundary, Aquila tunnels conventional Ethernet/IP packets over GNet, disassembled at ingress and reassembled and re-ordered at the egress of the Clique.

Unified fabric for legacy traffic and RMA/memory disaggregation. Aquila unifies low-latency communication primitives (RMA) alongside commodity primitives (IP) in a common fabric, to address the growing diversity of datacenter workloads [5, 42, 45]. A fabric delivering both high-performance and legacy connectivity avoids the pitfalls of a *bag-on-the-side* network and secondary NICs, reducing the cost of ownership and the toil related to the life cycle management of two separate networks. Managing a single network for availability, security, monitoring and upgrades

is challenging enough–managing separate networks for individual use cases introduces an extraordinarily high bar in any cost/benefit analysis. For efficient remote memory access and memory disaggregation alongside traditional protocols, we co-designed a Remote Memory Access protocol, 1RMA [51], to extend memory access across the Aquila Clique directly on GNet, instead of layering on top of IP.

Co-existing within the larger Clos-based software-defined datacenter network ecosystem. Typical datacenter networks [49] are based on a scalable Clos topology where aggregation blocks are connected via a spine switching layer; Aquila is designed to integrate into such a network via its Ethernet ports. A hierarchical Software Defined Network (SDN) control plane with a modular, micro-service architecture [18] manages and controls the various networking blocks within the datacenter.

Figure 2 describes the integration of Aquila in the broader datacenter network's dataplane and control plane ecosystem. The Aquila network block connects to the datacenter's spine switching layer via Ethernet links, akin to other aggregation blocks. The modular architecture of the datacenter network realizes a hybrid topology, i.e., a Dragonfly network integrated as a block within a larger Clos topology, a first of its kind to the best of our knowledge.

For the control plane, we adapted an SDN controller to configure, manage and program TiNs inband via a thin on-box firmware running on the TiN CPU (Figure 1). The Aquila SDN controller, similar to the SDN controllers of other aggregation blocks, interacts with each of four central Inter-Block Routing Controllers (IBR-C) (Figure 2) to enable communication with other aggregation blocks as well as with networks external to the datacenter.

Cliques as the basis for hosting tightly-coupled applications. To exploit the tightly coupled, low latency communication enabled by the Aquila Clique, we adapted the job scheduler [53] to be aware of Clique locality. High bandwidth or latency sensitive jobs could optionally be scheduled on host machines within a Clique, while other jobs could still be bin-packed across blocks, regardless of locality.

3 HARDWARE DESIGN

In this section we relate how the key design goals drove the hardware design. In summary:

- **Low latency** objectives drove the selection of a shallow-buffered cell-switched GNet fabric. §3.1 details the design of the GNet switch and link-level protocol.
- **Cost-effectiveness** goals led to the choice of an integrated switch and NIC chip, TiN, as well as a direct topology such as the Dragonfly. §3.2 outlines the rationale for selecting the Dragonfly and the impact of this choice on the design.
- **Shared fabric for both IP traffic and low-latency RMA.** §3.3 describes how IP packets traverse the GNet fabric, and §3.4 details the co-design aspects of the 1RMA protocol with the GNet fabric.
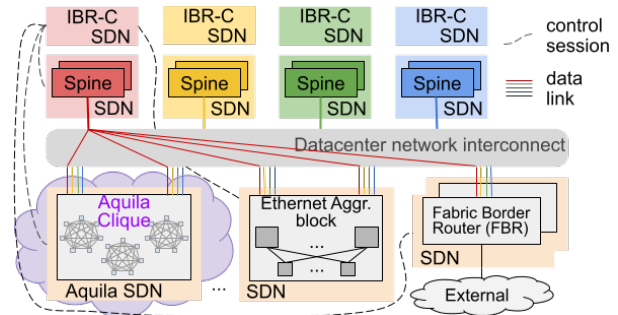
Figure 2: Aquila Clique integrated into the broader datacenter network and SDN ecosystem co-existing with other Ethernet aggregation blocks. Topologically, the Aquila block connects to the Clos-based datacenter spines akin to the Ethernet based blocks. In the control plane, the Aquila SDN controller, similar to controllers of other blocks, interacts with each of the four *sharded* Inter Block Routing controllers (IBR-C) to enable cross-block routing.

3.1 GNet Switch and Links

The cell switch. The switching capability of the TiN chip is provided by a 50-port cell switch optimized for low latency. The maximum cell size of 160 bytes was chosen to keep the serialization latency on 25G links small (~50ns). 32 ports are external-facing GNet ports (of which 24 are pod-local and 8 are inter-pod ports). The remaining 18 ports are intra-chip, for cells transmitted and received by the various traffic endpoints (e.g., IP and 1RMA). The fall through latency of the core cell switch is 20ns and the total per hop latency without Forward Error Correction (FEC) is 40ns. GNet links support 32 Virtual Channels (VCs [14]) - FIFO queues used for deadlock-avoidance and QoS. VCs are used for deadlock-free routing, for differentiation between classes of service, and to separate solicited and unsolicited traffic. A centralized arbiter implements a variant of the iSLIP arbitration protocol [38], supporting one arbitration request per VC per port, ensuring that no VC or port is starved of throughput. To support variable cell sizes, we modified iSLIP such that the ingress and egress ports communicate a "busy" signal to the crossbar arbiter. A "busy" indicates that the ports are transferring a cell across the crossbar. The arbiter takes this into account when it evaluates pending requests for the next request-grant-accept cycle. Quality of Service (QoS) between VCs is implemented in the output buffer and supports both weighted round robin and strict priority. Each VC has its own input FIFO space protected by a reliable credit mechanism, similar to that used in PCI Express. A shared buffer, shared credit scheme was considered to save memory, but for the relatively short links required for Aquila the simplicity and complete QoS isolation of independent FIFOs was preferred.

GNet link level protocol. GNet links support cells between 16 bytes and 160 bytes in size, with frequent reverse flow control traffic. The use of variable length cells gives very high protocol efficiency (e.g., 1RMA requests are small) on the wire even after the additional control traffic for admission control. Every GNet cell has a common routing header of 8 bytes that contains the 16 bit source and destination GNet ad-

dresses, the cell length and type, the VC, a decrementing hop count, an 8 bit header CRC and a Trace Enable bit. To enable efficient transmission of GNet cells, a custom 66/64 bit Physical Coding Sublayer (PCS) was developed that minimizes the cell delineation overhead and allows the reverse flow control traffic to be sent as very compact ordered sets. Control ordered sets are used for: (1) Start of cell delineation, (2) Flow control, (3) TimeSync (§B), (4) Management ordered sets (MOS), and (5) Phy up/down control and fault detection.

3.2 The Dragonfly cell fabric

We selected the Dragonfly topology to manage the cost of optical links, shown in Figure 1. The Dragonfly cell fabric is implemented using two types of GNet links: 24 *local* links per TiN that fully connect TiNs within the pod, and 8 *global* links per TiN that connect between pods, up to 100m apart on the datacenter floor. The local links are implemented as single-lane, 28 Gbps copper backplane connections. The global links are optical and use specially developed low cost GNet optical modules. Noise on global links is mitigated with FEC, incurring a 30ns per-hop latency penalty, and a 6% bandwidth overhead. Local links operate without FEC for the lowest latency at acceptable margins. Both local and global links ultimately implement the same link level protocol. Due to the hierarchical nature of the Dragonfly topology, GNet addresses have three components: *pod id*, *TiN id* and *endpoint id*. Endpoints represent protocol engines (IP, 1RMA, and CPU) detailed later.

Deadlock avoidance. We implement deadlock avoidance in our Dragonfly topology using a combination of turn rules and VCs. With our budget of 32 VCs, it is desirable to minimize the number of VCs used for deadlock avoidance. In the implemented routing scheme, we employ turn rules similar to the *parity-sign* approach in [20] within a pod for deadlock-free intra-pod routing. The VC is incremented when moving from a global link to a local link [30], requiring a total of 3 VCs used for deadlock avoidance in the worst fault-free route, that of a non-minimal route via an intermediate pod. Accounting for 10 traffic classes, each with 3 routing VCs, a further two VCs are available as *escape VCs* in certain dynamic failure avoidance scenarios.

Adaptive routing. The majority of traffic routes adaptively to achieve both high throughput and the lowest latency on Aquila's Dragonfly network. TiN implements locally adaptive routing [30, 48], a scheme that makes adaptive routing decisions based on available information at a GNet switch, in particular, the per-VC output queue lengths at each port. These queue depths reflect nearby congestion because of GNet's link-level flow control and shallow buffering. Link failures manifest similarly, which also allows the adaptive routing algorithm to route around failed links until the SDN routing engine removes the entries for links which have lost connectivity.

The adaptive routing implementation selects two minimal routes at random from eight supplied by the routing tables, and

also considers three non-minimal routes from 24 non-minimal route candidates. The five candidate routes are evaluated using a weighted comparison that favors the minimal routes. Random choices (rather than 24-way comparison) allow us to avoid flocking, having coordinated adaptive routing decisions, and moving congestion from one place to another [40]. Other routing modes are enabled by constraining the routing to minimal routes only, or by forcing deterministic choice of route using a hash of the source and destination addresses. These constraints yield Aquila's four principal routing modes: Fully Adaptive, Minimal Adaptive, Deterministic and Minimal Deterministic. The deterministic routing modes are used for cell types requiring ordering. The cell switch uses table-based routing because of the need to handle failures and upgrades using SDN routing described in §4.2, as well as flexibility for other topologies.

3.3 IP Traffic

Host IP traffic is sent and received by a conventional 100 Gbps NIC capable of supporting multi-host operation for up to two independent hosts. The option of multi-host capability was considered important in order to give a degree of flexibility in bandwidth per machine allocation. There are two other sources of IP traffic on the TiN chip: the 100 Gbps external Ethernet port for connectivity outside the Aquila fabric, and a low bandwidth port to the embedded management processor. Traffic from all IP sources is handled in the same way: the packet processing pipeline performs IP routing and cellification, i.e., splitting the IP packet into GNet cells and traversing them to the final destination, using Aquila's IP over GNet protocol.

Packet processing logic. Each IP packet passing over the GNet fabric goes through the input packet processing and output packet processing blocks once only. Effectively, the entire GNet Clique acts as a single stage IP packet switch. The input packet processing pipeline handles:

- L3 to GNet L2 address translation (either one-to-one or WCMP [55]);
- Selectively punting some packets to the embedded control processor;
- Input buffer QoS.

L2 Ethernet MAC addresses are stripped from inbound packets after processing; packet transfer over the GNet cell network is for IPv4/IPv6 only. Non-IP packets such as ARP may be either encapsulated or punted to the embedded control processor, consistent with the requirements of our SDN control plane (§4).

IP over GNet protocol. IP traffic traverses Aquila by means of the GNet upper layer protocol, shown in Figure 3. Each IP packet sent over the cell fabric issues a *Request To Send* (RTS), and awaits a *Clear to Send* (CTS) handshake before any data is transmitted. These are sent as 16 byte GNet cells to minimize the bandwidth overhead. The handshake protocol performs three functions:
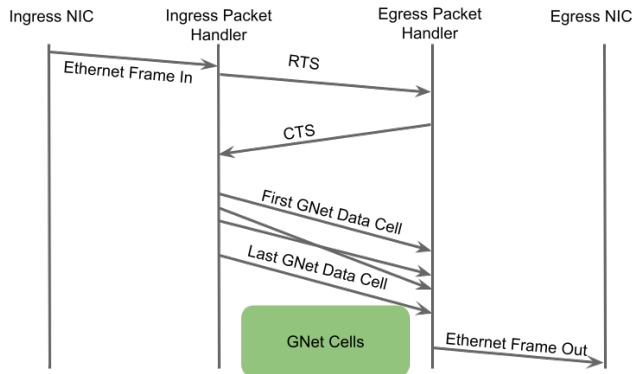
Figure 3: Cellification: IP Packets are split into multiple GNet data cells that are only admitted into the GNet fabric when the ingress TiN receives a CTS. In the egress packet handler, cells are reassembled into packets, respecting their original transmission order, and sent to the NIC (all within the TiN chip).

- It implements solicitation for IP packets by only allowing data cells onto the GNet network when the destination end point has signalled it has sufficient input bandwidth and buffer space to receive them.
- It allocates hardware resources at the destination, e.g., cell to packet reassembly buffers, before any data cells are transmitted so that there is always a guaranteed reassembly space.
- RTS arrival defines inter-packet order. While data cells route adaptively and potentially arrive out of order at their destination, RTS ordering ensures that original transmission order can be reconstructed at the receiving side.

Packets which have passed through the packet processing pipeline and have a valid GNet L2 address are stored in the packet ingress buffers. An RTS is generated immediately; in fact, for long packets the RTS can be issued before the whole packet is received. The RTS itself consists of 8 bytes of routing header and a further 8 bytes of payload that includes the IP packet length, Class of Service (CoS), the packet's location in the ingress buffer, and an indicator of ingress buffer usage. Compactness is important because RTS cells are unsolicited and can still lead to incast. However, considering that an average packet is >1Kbytes, an incast of RTS cells represents a reduction of incast volume in the network by a factor of 64.

RTS cells are carried on their own VCs, allowing them to be sent at high priority and also maintain isolation between solicited and unsolicited cells. RTS VCs are routed deterministically over the GNet fabric, using a path selected by a hash of the flow-invariant fields of the cell, ensuring that the RTS cells for a given IP flow are received in the order they were sent. The RTS cells are received into FIFO queues at the packet egress. Packet data transfer is initiated by the egress-side by sending a CTS back to the appropriate ingress port. Along with the 8-byte routing header, the CTS carries a pointer to the packet in the ingress buffer (copied from the RTS) and a pointer to the allocated location in the egress cell-to-packet

reassembly buffer. CTS cells are issued by the CTS scheduler, which tracks the availability of egress reassembly buffer capacity, only issuing a CTS when there is space available to reassemble cells into packets.

When a CTS is received back at the packet source, the packet in question is pulled from the ingress buffer as a series of data-only cells, which are then transmitted across the fabric. Data cells can take many different routes (adaptively) through the fabric, and data cells may arrive in any order at the final destination. Cells are reassembled into packets in the egress buffer at the destination. The sizing of the egress buffer is determined by the bandwidth delay product of the output port bandwidth and the cell fabric round trip delay, plus an allowance for packet reordering delays. Packets do not experience significant queuing in the egress buffers, which are primarily for reassembly, so the egress buffers are significantly smaller than the ingress buffers.

In order to maintain packet order within flows, when a CTS is issued by the scheduler, the packet descriptor is registered with packet reordering logic respecting RTS arrival order. A packet is transmittable at egress after receipt of all its data cells, but transmittable packets are held until all packets in the same flow that were ahead of it in CTS issue order have been successfully forwarded to the NIC.

A significant benefit of the RTS/CTS scheme is that the RTS queues have a local view of all the requested packet demand for that destination port from the entire GNet fabric, while the packet *data* remains queued in the ingress buffers. In the presence of severe incast, packets can be discarded while conserving fabric bandwidth, i.e., without packet data traversing the cell fabric. The egress side can choose to drop a packet by issuing a variant of CTS (a *Clear To Drop*, CTD), which pulls the packet from the ingress buffer and discards it. A CTD is sent when an RTS is received at an RTS queue whose depth exceeds a given threshold. The RTS queue's depth also provides the signal for Explicit Congestion Notification (ECN) marking; if the RTS queue exceeds the marking threshold when the packet has been reassembled and is ready to send, ECN is applied.

QoS Support for IP. There are separate RTS queues for each independent port and class of service, with a total of 32 RTS queues supporting eight CoS on four independent ports - one port for the external Ethernet MAC, two for the dual-host NIC, and one for the control processor. CTS cells are issued by the CTS scheduler at the packet's destination which allocates bandwidth between the 32 RTS queues, implementing per-IP-packet QoS between the respective queues. The CTS scheduler may throttle traffic into the egress buffers by limiting CTS issues according to a window of outstanding packet fetches, which can be adjusted to minimize the queuing of data cells within the cell fabric. The scheduler does not attempt to implement bandwidth fairness between sources since all the sources to a given destination port share the same RTS queue.

3.4   1RMA

To deliver the low-latency capabilities of the Aquila Clique directly to distributed systems programmers, we built an implementation of 1RMA [51] into the TiN chip. 1RMA is an RMA protocol that offers unordered, segmented, solicited remote memory access primitives (read, write, and atomics) to on-host software—tenets that match precisely those of GNet packet transfer governed by RTS/CTS.

Such alignment is not merely coincidental; we co-designed Aquila and 1RMA's GNet-based protocol. Rather than simply layering the 1RMA protocol messages above the packet layer, we instead express 1RMA protocol exchanges as first class cell types in GNet—alongside RTS and CTS, rather than atop—and ensure that they obey similar end-to-end solicitation rules as they share the Aquila fabric. The advantage of co-design is significant latency savings: while a UDP/IP or TCP/IP round-trip on Aquila incurs *six* GNet half-round-trips on its critical path (RTS, CTS, data, in each direction), a 1RMA read operation incurs only *two*, shaving precious microseconds from user-facing latency.

We realized protocol co-design by encoding 1RMA *read requests* entirely within GNet framing. Fundamentally, read requests initiate data transfer from receivers to senders, i.e., such requests intrinsically already are solicitations, expressed at the transport layer. GNet also builds on solicitation, but at the L2 layer. The key insight is to express both the GNet (L2) and 1RMA (L4) solicitation behaviors in a single cell type, *Req*. Since Req cells solicit data movement in the reverse direction, GNet handles Req similarly to CTS; the main differences arise from cell size, as Req fully encodes a read request (host address, memory identifiers, HMAC, etc.), yielding a cell 3x larger than CTS at 48B. Req is otherwise behaviorally similar to CTS, in that it can be freely reordered without violating assumptions of the protocol layer above. Because 1RMA is highly tolerant of out-of-order delivery, Req is intrinsically compatible with Aquila's adaptive routing.

We also leverage 1RMA's close coupling to host-facing PCIe to encode response cells, *Resp*. 1RMA NICs send each individual PCIe read completion payload as a distinct protocol response, a hardware simplification that avoids response coalescing logic, buffering, and overheads in the NIC. To facilitate this behavior in GNet, Resp cells are sized to handle the most common PCIe completion sizes we observe from the host root complexes. Like Req, Resp can be freely reordered and routed adaptively, and the initiating 1RMA NIC lands the individual response segments in arrival order in destination host memory, since there is no need to restore overall inter- or intra-request response ordering.

Lastly, to isolate latency-critical 1RMA traffic from less sensitive IP flows, we map roughly half of GNet's virtual channels to carry low-latency protocol messages, which 1RMA shares with low-latency IP traffic flows. Because IP traffic is cellified, 1RMA responses do not queue behind bulk transfers from competing flows. In all, 1RMA on Aquila de-

livers near-flat lookup latency—even under load from conventional traffic—to approximately 864TB of DRAM inside of 4us end-to-end. Aquila traversal accounts for a mere 2.5-3us; the remaining time is attributable to PCIe latency contributions.

3.5   Embedded control processor

The TiN chip has an embedded control processor (ECP) to handle all switch side control and monitoring actions. Cost of silicon exerts pressure to make the ECP as simple as possible, as it is replicated in each TiN chip. Where a typical control processor for a ToR might be a multicore, 64-bit processor with 8-16 GB of memory, TiN's ECP is a 32-bit ARM Cortex M7 processor with a mere 2 MB of SRAM.

In order to bootstrap the embedded control processor before the GNet logic has been fully initialized (§4.4), a low bandwidth but reliable in-band control path is implemented over the GNet fabric using the management ordered set (MOS). Each MOS 64 bit word allows 6 bytes of data to be transferred between directly connected TiN chips, irrespective of whether the GNet link layer is up. We layer a robust packet implementation, PMOS, above the MOS primitive to carry debug and bootstrap traffic.

3.6   Putting it all together

Figure 4 plots the overall structure of the TiN chip:
- The cell switch (the building block for the cell fabric);
- A conventional IP host interface (NIC);
- An external-facing Ethernet MAC for connectivity to outside networks;
- A 1RMA host interface that supports direct protocols across the cell fabric;
- IP packet-to-cell (ingress) and cell-to-IP packet (egress) logic;
- The embedded control processor, acting as the local agent for the SDN control software.
  The device has the following interfaces:
- Two x16 PCIe gen 3 interfaces giving 256 Gbps connectivity to one host or 128 Gbps to each of two hosts;
- Thirty-two 28 Gbps, single-lane GNet links used to construct the low latency cell fabric;
- A single 100 Gbps Ethernet interface which connects to the wider datacenter network (DCN).

Approximately 50% of the TiN silicon area is used to implement host interface or NIC functions, and 50% for switching functions.

4   SOFTWARE-DEFINED NETWORK

As alluded to in §3, the integration of switch and NIC in a single chip leads to substantial replication of the management subsystem across all TiNs in an Aquila Clique. To keep the Aquila Clique cost-effective, the management subsystem for a TiN ASIC was kept simple – a 32-bit ARM Cortex M7 processor with a modest 2MB of SRAM and no dedicated management Ethernet port. Consequently, much of the routing
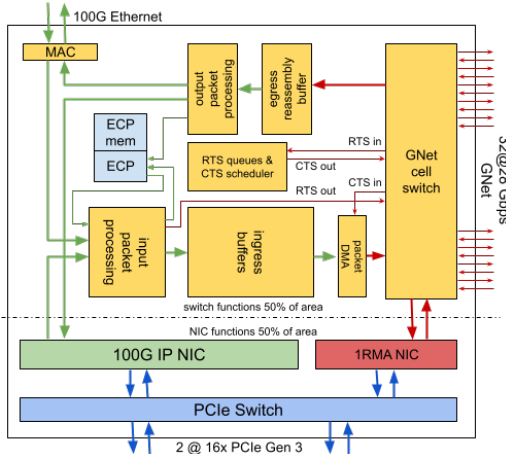
Figure 4: Aquila Chip Architecture showing GNet, IP, Embedded Control Processor (ECP), 1RMA and PCIe components.
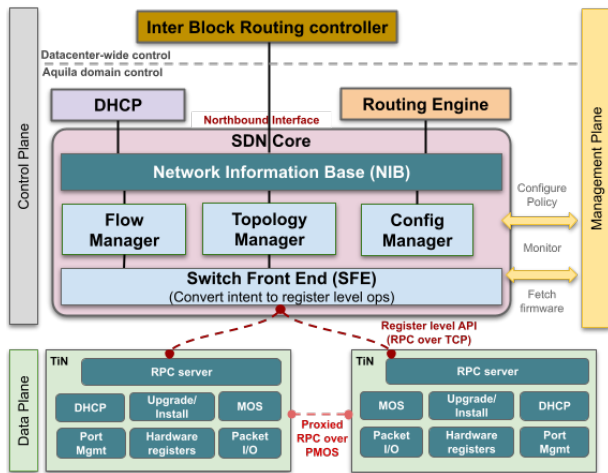

Figure 5: Overview of Aquila's SDN and firmware architecture.

computation and state needed to be offloaded from the switch to a logically centralized distributed controller which had to orchestrate the bootstrap, management and control of the fabric in-band. In this section, we describe how Aquila's software-defined network (SDN) control plane, along with its simplified firmware, was able to address the challenges of controlling and managing an Aquila Clique, specifically:

- Explosion of flow state in the SDN (§4.2).
- Switch firmware with constrained CPU/memory (§4.3).
- In-band bootstrap of the whole network (§4.4).

4.1 Control Architecture Overview

The Aquila controller is built on top of an SDN controller platform [18], a modular SDN control plane comprised of micro-services, and a central publisher/subscriber database called the Network Information Base (NIB). Multiple applications form an Aquila SDN constellation with redundant instances of each application deployed on separate control servers. The top half of Figure 5 details the Aquila SDN controller applications.

The routing application for the controller, Routing Engine (RE), computes the routing solution for the Aquila network

block in reaction to changes of topology states and external reachability. RE writes the solution in the form of flows and groups similar to OpenFlow [39] to the NIB in sequenced batches for hit-less routing state transition. Separately, the Inter-Block Routing controller (IBR), an application in a data center-wide SDN control domain, computes the routing solution for traffic between various network blocks and provides Aquila's RE with the egress paths to reach destinations external to the Aquila Clique.

On receiving routing updates from the NIB, Flow Manager (FM) sorts the flow and group programming operations. For instance, a flow is installed only after its referenced group is installed for hit-less transition, before sending them to the Switch Front-End application (SFE) via RPC. SFE programs the flows and groups to TiN switches converting between flows/groups and hardware register values and completes the RPC with the programming results. Then FM writes the results back to the NIB for RE to consume.

4.2 Handling routing state

The large number of GNet endpoints in the fabric and the per-port GNet routing table in the TiN switch result in much larger routing state than non-Aquila blocks, which increases both CPU and memory demand in the SDN system. Aquila routing introduced scaling challenges for both IP and GNet flows.

IP flows. A network comprised of 1152 hosts and 576 management CPUs, addressable via both IPv4 and IPv6, calls for approximately 1.9 million flows, each with a single output port. Leveraging the observation that all of these flows are from a small number of subnets, we introduced a new *indexed* group representation, where the index of a port in the group corresponds to the same index in the subnet, which in turn reduces the number of flows by a factor of 576 (the number of TiNs in a fabric).

GNet flows. As seen in §3, flow controlled GNet requires per-input port, per-virtual channel flows which leads to an explosion in state for a switch with 50 ports. A naive implementation leads to almost 5 million flows. To accommodate such a large scale, we exploited the significant similarity in the routes. For example, all terminal ports described in §3.1 use the same route, and are represented only once in the NIB. Similarly, the deadlock avoidance turn rules define a similar role for each intra-pod port in the TiN chip. Further, all inter-pod ports behave the same. We introduced six *port classes* – denoting equivalence classes of ports with respect to routing rules – reducing the number of flows to approximately 700k. GNet flows use port-classes as both matching fields and output actions. On receiving a GNet flow using a port-class, SFE expands it to flows targeting each member port's GNet flow table, and then prunes improper member ports from the output, e.g., to avoid sending traffic back to the source. The resulting flows are then programmed in the switch.

Even with these optimizations in place, the rest of the SDN system needed more modifications:

- Despite the port-class optimization, the number of flows in the NIB was still about 10x more than non-Aquila network blocks. To compensate for the memory increase, the NIB's pub-sub interface was changed to keep state in compressed format and decompressed only when necessary.
- The SRAM available in the TiN switch is not large enough to hold a snapshot of all routing state. SFE has the capability to rate limit the hardware programming operations to avoid the memory on the switch from overflowing. The RPC interface between SFE and switches is designed in such a way that the largest RPC can fit in memory and only one outstanding RPC is allowed at a time.

4.3 Switch Firmware with limited state

The switch firmware (see lower half of Figure 5) runs on an ARM Cortex M7 CPU integrated into the TiN switch chip. Due to physical size and cost limitations, the firmware has only 2MB of on chip SRAM available. Therefore, it is built on the FreeRTOS [1] and lwIP [2] open source libraries to fit within the space constraints. The firmware is implemented in approximately 100k lines of C and C++.

We explicitly decided that the firmware is not responsible for fully configuring the TiN chip. At power on, the firmware brings up the GNet and Ethernet links and attempts DHCP over Ethernet. This enables the controller to connect early during initialization and finish the necessary configuration to allow the TiN chip to start passing traffic (for details see §4.4).

The programming API exposed by the firmware is low-level and allows the SDN controller to directly access hardware registers. The API is generated from the hardware register description and permits the SDN controller code to use symbolic names of the chip registers for convenience. Statistics and counters from the TiN chip can also be reported using the low level API. The SDN controller is able to configure a set of registers that should be periodically reported by the firmware. One of the programming API sets up ARP/NDv6 responses in reaction to requests from the attached machines so that the IP-to-MAC resolution could function properly even if the firmware loses connection to the SDN controller.

The firmware supports *Non-Stop Forwarding* (NSF) reboots to minimize disruption caused by upgrades and unexpected software errors. During reboot the firmware avoids changing any configuration that might impact traffic. Since the inband connectivity is not disrupted, the controller is able to quickly reconnect after a reboot without going through the bootstrap process. The implementation of NSF reboot was simplified due to the register level API since there is no need to save and restore state information, because the TiN chip maintains all the controller visible state during reboot.

While the firmware itself is stateless, the TiN chip and SDN controller are not. After any loss of connection between the firmware and SDN controller a process of reconciliation has to be initiated to resolve any differences between the hardware registers and the SDN controller intent. These differences can occur if any commands were lost when the connection failed.

4.4 In-band Control and Bootstrap

A key challenge in Aquila's SDN control was that the control channel from the SDN controller to the Aquila switches is in-band. This means that the controller needs to communicate with the management CPU of a TiN before it can program the routing tables of the TiN. During bootstrap, the controller sets up TCP connections in-band over the datacenter network to all TiNs in the Clique in iterative "waves", configuring and programming routing tables as it gains control of TiNs in each subsequent wave.

Figure 6 shows *k* Aquila pods connected via intra-pod copper GNet links as well as inter-pod optical GNet links. Some TiNs (e.g., TiN 1, TiN 3 in Pod 1 and Pod k) are also connected to the spine layer of the datacenter via Ethernet datacenter network (DCN) links. We refer to these TiNs as *DCN-connected*. The Aquila SDN controller—running on external control servers—is initially reachable only over the DCN links. The TiN firmware sends DHCP discover messages over the DCN links if available. These DHCP messages are relayed by the spine switches to the DHCP server, which then assigns an IP address to the TiN management CPU based on the TiN MAC address.

The Aquila controller has records of the IP addresses intended for each TiN's CPU from its own configuration. The controller continually attempts to connect to each TiN CPU via TCP session using its assigned IP address and a well known L4 port number. Once the IP address is known to a TiN's firmware, a controller message destined to that IP is trapped by an ACL rule installed by the firmware and reaches the firmware. The response Ack is sent out the same interface the packet came in from thus enabling a TCP connection between the controller and the switch CPU of the DCN-connected TiNs. The controller can then configure the DCN-connected TiN and program its routing tables.

Once the controller establishes a TCP session with a DCN-connected TiN, it uses that TiN as a *proxy TiN* (e.g., Pod 1, TiN 3) to bootstrap a directly connected *target TiN* (e.g., Pod 1, TiN 2) using the point-to-point low bandwidth Packet Management Ordered Sets (PMOS) protocol (§3.5) between TiN CPUs. A target TiN—not yet configured with its IP address—also sends DHCP discovery messages over MOS over all GNet links, which are trapped by the proxy TiN and sent over its own session to the controller. The controller in turn relays the discover message to the DHCP server, and likewise relays the DHCP response, so that a target TiN learns of its assigned IP address indirectly. The controller proceeds to configure and program the routing tables in the target TiN via the proxy TiN over PMoS. After enough routing state is programmed, the controller can establish a TCP connection to the target TiN via the GNet routing pipeline and the proxy TiN (Pod 1, TiN 3) can then be used in turn to bootstrap yet another target TiN (e.g., Pod 1, TiN 4). Once a TCP session is established
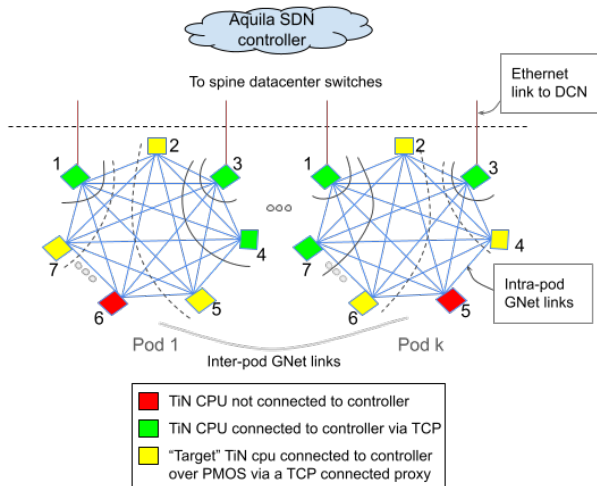
Figure 6: Inband bootstrap. The Aquila SDN controller bootstraps the TiNs inband in "waves" originating from the TiNs that are directly connected to the DCN.

with this new target TiN, it too can be used as a proxy to bootstrap a directly connected TiN (e.g., Pod 1, TiN 5), and so on.

DCN-connected TiNs typically bootstrap faster than the target TiNs, which are configured over the slower PMOS protocol leading to a distribution of bootstrap times ranging from 3 minutes to 48 minutes. Several of the waves of bootstrap occur simultaneously, resulting in a bring-up time of approximately 2.5 hours for a full-sized Clique.

5 EXPERIMENTAL RESULTS

We present a set of results examining key aspects of the Aquila network, including its data plane performance as well as its impact on application metrics.

5.1 Data Plane Performance

Aquila's data plane performance was evaluated in a prototype Aquila testbed comprised of 576 TiNs. We used 500 host machines. Two hosts share a NIC unless otherwise specified. We used two workloads, both of which run with delay-based congestion control [33]:

1. *UR*: An IP traffic generator based on a user space micro-kernel, *Pony Express* [37], that generates a Uniform Random traffic pattern with Poisson arrival.
2. *CliqueMap*: A key-value store [50] that uses Remote Memory Accesses (RMA) via either Pony Express or 1RMA.

For our evaluation, we used three metrics:

1. *IP Fabric RTT (μs)*: We used NIC hardware timestamps to measure Aquila fabric RTT, excluding processing and ack-coalescing delays on the remote host. This is a true measure of the transmission and queuing delay inside the Aquila fabric, both for GNet and IP components.
2. *1RMA Total Execution Latency (μs)*: the time from when the RMA command is submitted to the hardware until the hardware issues the completion for that command.

This metric measures more than queuing and transmission delay in the fabric, as it includes the PCIe transaction delay on the remote side.

3. *Achieved throughput* of the network in Gbps (averaged over 30 seconds).

Latency Under Load. We examine the latency of both IP and 1RMA traffic under load. We used a CliqueMap client benchmark that issues lookups of 4 KB-sized values using RMA. By varying the QPS of the CliqueMap client on the 500 hosts, we changed the offered load per machine in a traffic pattern akin to Uniform Random. Figure 7 plots fabric RTT against offered load. It shows that the fabric latency remains under 40 μs, even when the network is close to the per machine NIC line rate of 50Gbps and it is sub-20 μs at 70% load.

1RMA is co-designed with GNet (§3.4) and Figure 8 shows that this co-design paid off with total execution time below 10 μs even under high load for 4 KB RMA reads that are generated using 500 CliqueMap clients to read from 500 CliqueMap backends.

1RMA Latency Isolation. Aquila is a unified network shared by low latency 1RMA traffic and regular IP traffic which may be latency insensitive. In our next evaluation, we show that Aquila delivers latency sensitive traffic with low tail latency despite sharing the network with IP traffic. To this end, we compare the latency of latency-sensitive traffic with and without background IP traffic in both Aquila and a conventional Ethernet network.

For the Ethernet network, we employ standard QoS techniques to isolate low-latency (or important) traffic from bulk throughput oriented traffic. We run 200 instances of CliqueMap lookups of 4 KB values at 10,000 QPS on a higher priority QoS class (H) and a UR traffic pattern with 64 KB messages with average load of 10 Gbps on a lower priority class (L). The relative egress scheduling priority between H and L classes is 8:1. The orange and cyan bars in Figure 9 show that such QoS-based schemes provide reasonable isolation for the CliqueMap traffic from the bulk IP traffic, leading to a modest increase in queuing in the fabric RTT for HiPri CliqueMap traffic, albeit with a high baseline latency.

Repeating the same experiment using 1RMA as a transport, we can see that 1RMA on Aquila offers a much lower baseline (less than 5 μs median and tail latency) despite sharing the same GNet fabric with IP traffic. High priority 1RMA traffic uses different virtual channels than low-priority Pony Express IP traffic and thus is nearly unaffected by adding the bulk traffic (blue and red bars). Even when low priority 1RMA traffic shares the virtual channels with the bulk IP traffic (yellow and green bars), the overall latency is slightly higher than 10 μs but still lower than Pony Express traffic on Ethernet networks (orange and cyan bars).

Effect of Burst Size. One of the lessons we learned in Aquila is the phenomenon of *self-congestion*. The IP network in Aquila has an injection rate of 100 Gbps per TiN while
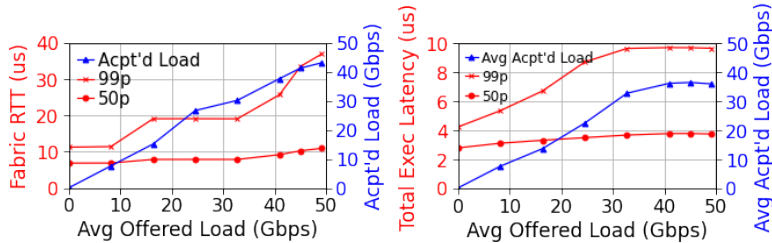
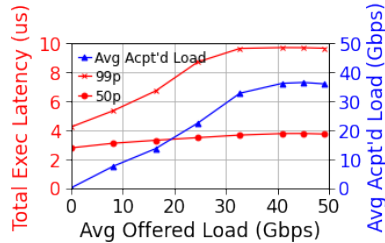**Figure 7:** IP Latency vs. Load: Fabric queuing remains low under load.

Figure 8: RMA read latency under varied 1RMA Load.
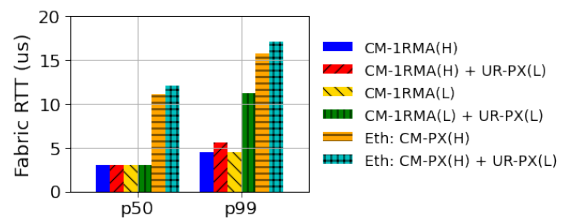

Figure 9: 1RMA Isolation: Aquila provides low latency for 1RMA traffic, even when sharing the network with IP (H = High Priority, L = Low Priority, CM = CliqueMap, PX = Pony Express)
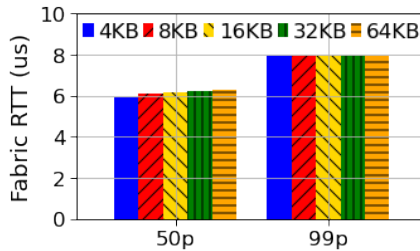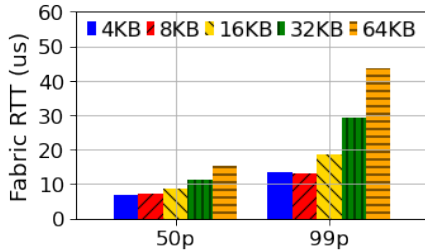

Figure 10: Effect of burstiness on queuing in a full sized Aquila (left) and a half sized Aquila (right). By keeping the injection rate constant and varying message size, we can see the effect of burstiness on queuing latency.
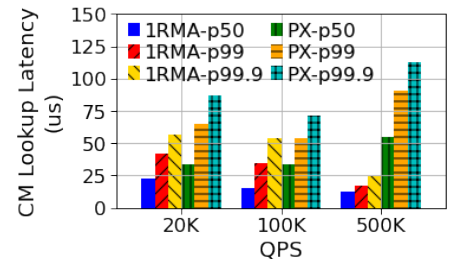
Figure 11: Effect of a cell-based RMA protocol on end-to-end CliqueMap lookup latency.

the aggregate bandwidth along the minimal paths between two pods in the full scale Aquila topology is limited to 50 Gbps. This leads to cells taking non-minimal routes even if the overall injection rate is well bellow the link rate due to bursts. We show this effect by keeping the injection rate of a point-to-point traffic at 0.6 Gbps but varying the message size of the RPC using Pony Express. Varying the message size only affects the burstiness of the injection. Figure 10 shows that as we increase message size, the tail fabric latency increases past 40 μs. However, repeating the same experiment in a half-scale version of Aquila where we have matching inter-pod bandwidth to the IP injection rate from each TiN, we see no effect of message size on queuing in the fabric. Provisioning higher minimal path bandwidth trades off better performance under bursty traffic conditions in exchange for a smaller maximum scale of the topology.

5.2 Application Impact

In order to see application impact, we compare CliqueMap lookup (of objects with 4 KB size) latency using 1RMA and Pony Express as a transport for RMAs on the Aquila network. We use O(100) backends and clients and vary queries per-second from each client. Figure 11 shows how 1RMA on Aquila cuts the median and tail latency by 50% at low QPS and by more than 300% at high QPS. As with prior work [51], higher load levels with 1RMA deliver *lower* latencies, as individual servers may dwell in low-power states at low load.

6 DISCUSSION

While our approach to Aquila's design enabled us to develop a unified low latency network fabric for datacenter networks, there were a number of challenges that we had to overcome. We highlight some of the key challenges next.

Single chip part and direct connect topology. While the single chip design delivered a sustainable development model

with a modest sized team and cost efficiency for the Aquila network, the approach had a couple of key implications on the architecture and deployment. First, the single part implied that we had to deploy Aquila as a direct connect network topology because an indirect topology (such as a Clos network) was infeasible with TiN chips. While not a drawback by itself, a direct connect topology is not conducive to incremental deployment. Secondly, the evolution of the NIC and the switch architectures were coupled together from a multi-generational roadmap standpoint.

For simplicity, we designed the Aquila Clique as a homogeneous unit of deployment without an intent to mix hardware from different generations. Moreover, the networking footprint for the entire Clique (up to 24 racks housing all TiN cards as well as the networking fiber) was designed to be deployed up front and host machines could be incrementally populated on demand. With an indirect topology, a small number of network racks (e.g., 4) could be pre-deployed with server racks deployed incrementally. With a direct topology, all server racks (potentially without servers) had to be pre-deployed. Further, for a given optical technology, an indirect topology supports more deployment flexibility: all server racks need only be within a (say) 100m radius of the network racks. With our direct topology, care had to be taken to lay out the rack footprint such that the GNet fiber length between all rack pairs was within the budget of (say) 100m.

Self congestion due to thin minimal path links. The scale of a Dragonfly topology can be increased until we have only a single global link between each pair of pods. However, a mismatch between the injection bandwidth from a TiN and the pod-to-pod bandwidth leads to *self-congestion* where, even at low loads and especially for large MTU packets, some cells may be routed minimally while others may traverse non-minimal paths. As a result, there is some vari-

ance in latency introduced due to cell and packet reassembly even for point-to-point flows at low average loads.

For our initial Aquila prototype, we chose a Clique size of 576 TiNs where the pod-to-pod bandwidth was 2x25Gbps which was 1/4 the maximum injection bandwidth of 200Gbps for each TiN, a balance between Clique scale and self-congestion in the Dragonfly configuration. Further, we tuned adaptive routing to switch from minimal to non-minimal paths to reduce the variance due to self-congestion.

Overhead of cell switching and solicitation. Cell switching and solicitation are key features in Aquila for achieving predictable, low network latency. Switching GNet cells comes with an overhead of approximately 5% due to an 8 byte GNet header for each 160 byte GNet cell. The RTS/CTS solicitation for each IP packet incurs a latency overhead of an extra round trip through the network though the RTS/CTS cells get high priority through the GNet network and the overhead is further mitigated for packets with large MTU. We considered both these overheads acceptable in exchange for low tail latency even at high injected loads. Considering a larger GNet cell size as well as the ability to not incur solicitation overhead at low loads are techniques we are investigating to further mitigate these overheads.

Debugging a cell switched network. Since the Aquila Clique is not an IP routed fabric internally, standard debug tools such as traceroute only show 1 hop through the entire Aquila fabric. To debug data blackholes in Aquila, we implemented a *cell tracing* capability in TiN. Cells that are marked with a bit are sampled by each TiN in the cell's path and sent to a central collector over UDP. The collector can then stitch the path of the constituent cells of a packet and triangulate any mis-configured or faulty hardware.

Limited RAM on TiN and low level firmware API. To save cost and board space, we provisioned just 2MB of RAM for the firmware running on the TiN chip, which led us to a custom firmware implementation. Firmware development added significantly to the development effort, since many basic facilities had to be customized or re-implemented (e.g., logging, memory allocation, and flash storage).

The decision to expose a register level API to the SDN controller for programming the TiN chip had the benefit of shifting complexity away from the resource constrained firmware as well as simplifying the capability to upgrade firmware with Non-Stop Forwarding (NSF). It also meant that new features could be implemented without needing to roll out a new firmware version since all features of the hardware were exposed. A challenge with this approach was maintaining this interface across multiple hardware generations, since the SDN controller would need to be aware of the register level details of each chip.

For future designs, we are investigating adding more compute to the NIC so that it can be Linux based. Adding RaspberryPi equivalent compute to each NIC is likely to minimally increase the per unit cost relative to the expected gains in de-velopment velocity. Additionally, more compute will unblock the use of an API with a higher level of abstraction, such as P4 Runtime [24].

Legacy Applications Performance. While Aquila delivered significant application performance improvements (§5) for the co-designed case, such as CliqueMap with 1RMA, it did not have a significant positive impact on legacy applications. We observed that legacy application's tail latency is dominated by the host software stack, including thread wake up latency. Moreover, with IP software stacks, RTS queue length is governed by the host congestion control algorithms rather than the GNet fabric cell latency. Looking forward, we are shifting transport and network protocols to natively take advantage of future-looking hardware improvements, creating an interesting tension where the substantial software investment would likely not be a net positive until newly designed hardware is deployed across the majority of the fleet.

7 RELATED WORK

Topology and Cell-switching. Aquila uses a direct-connect topology, Dragonfly [30]. The Cray Cascade system [17] utilizes a Dragonfly topology as the basis for an HPC fabric. This design uses a high radix switch with 4 integrated host interfaces, using a proprietary packet format and virtual cut-through. The gateway to Ethernet networking requires processing nodes connected to both types of network. Aquila differs from this system (and from work on Flattened Butterflies [31] and HyperX [3]) by using the topology as a cell fabric, as opposed to a packet network with virtual cut-through. JellyFish [52] is a random-graph topology with its own challenges of deployment. Sirius [6] is a flat-topology with similar goals to Aquila but it utilizes optical circuit switching rather than cell or packet switching. Early ATM networks provided Ethernet-on-ATM [26]. More recently, Stardust [56] employed the idea of cells to give a higher effective switch radix by using single lane channels in the fabric.

Low latency networking protocols. Infiniband [10] implements an alternative networking stack to Ethernet/IP optimized for lower latency. This provides a flow controlled, lossless packet level protocol, a reliable transport implementation, and a complete set of messaging and RDMA operations. Although inter-operation with Ethernet networks for IP traffic can be implemented by gateway functions, Infiniband is commonly used as a dedicated HPC network. SRD [47] (Scalable Reliable Datagram) is an alternate transport protocol layered over IP datagrams that is used in conjunction with EFA (Elastic Fabric Adapter) by a Cloud computing provider to provide lower latency communications services for HPC applications. This has the advantage of being able to use standard Ethernet switches at some cost in minimum achievable latency. While Aquila uses cell-based adaptive routing, SRD uses source-based adaptive multi-pathing.

Congestion control. Solicitation is one of the key elements of GNet for controlling congestion in the GNet fabric. A

few recent congestion control schemes such as Homa [42], NDP [25], ExpressPass [11], pHost [19] and Stardust [56] use a receiver-driven solicitation scheme, similar to that of GNet, to avoid incast congestion and achieve low latency. Aquila's solicitation controls the transfer of IP packets from buffers at the GNet fabric edge and does not directly control the IP NIC. This means it can handle both gateway and host interface IP traffic, but it requires host-based congestion control [33] to cause the traffic sources to back off in the event of congestion.

Control-plane. Aquila's control plane was designed with a distributed software defined control-plane. Most of the previous SDN controllers, Onix [32], ONOS [9], Flowlog [43], Ravel [54] assume routing of IP traffic and rely on Open-Flow to program switches. Aquila's control plane introduces a lower-level communication protocol from a Switch Front-end module to control light embedded switch controllers. The table-based design in [18, 43, 54] allowed for extending routing and sequencing to support GNet flows in addition to IP flows.

8 CONCLUSION

In this paper we present Aquila, our first foray into tightly-coupled networks (Cliques) integrated within the datacenter networking ecosystem realizing Clique-scale resource disaggregation and predictable, low-latency communication. Our primary goal is to advocate for a new design architecture for datacenter networking around Cliques and to encourage new research and development in tightly-coupled networking in support of high-performance computing, ML training, and network disaggregation while simultaneously interoperating with traditional TCP/IP/Ethernet traffic at datacenter scale. We believe our experience, both positve and negative, with the Aquila prototype will set the foundation for future exploration in this space.

REFERENCES

[1] Freertos: Real-time operating system for microcontrollers. https://www.freertos.org/. Accessed: 2022-02-28.

[2] Lwip: A lightweight tcp/ip stack. https://savannah.nongnu.org/projects/lwip/. Accessed: 2022-02-28.

[3] Jung Ho Ahn, Nathan Binkert, Al Davis, Moray McLaren, and Robert S Schreiber. Hyperx: topology, routing, and packaging of efficient large-scale networks. In *2009 SC Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2009.

[4] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *SIGCOMM Comput. Commun. Rev.*, 38(4):63–74, August 2008.

[5] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. ACM.

[6] Hitesh Ballani, Paolo Costa, Raphael Behrendt, Daniel Cletheroe, Istvan Haller, Krzysztof Jozwik, Fotini Karinou, Sophie Lange, Kai Shi, Benn Thomsen, and Hugh Williams. Sirius: A flat datacenter network with nanosecond optical switching. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 782–797, New York, NY, USA, 2020. Association for Computing Machinery.

[7] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 155–168, 2017.

[8] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 328–341, 2016.

[9] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, et al. Onos: towards an open, distributed sdn os. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 1–6, 2014.

[10] Rajkumar Buyya, Toni Cortes, and Hai Jin. *An Introduction to the InfiniBand Architecture*, pages 616–632. 2002.

[11] Inho Cho, Keon Jang, and Dongsu Han. Credit-Scheduled Delay-Bounded Congestion Control for Datacenters. In *Proceedings of the ACM SIGCOMM 2017 Conference*, SIGCOMM '17, pages 239–252, New York, NY, USA, 2017. ACM.

[12] David D. Clark, John Wroclawski, Karen R. Sollins, and Robert Braden. Tussle in cyberspace: Defining tomorrow's internet. *IEEE/ACM Trans. Netw.*, 13(3):462–475, June 2005.

[13] Paolo Costa, Hitesh Ballani, Kaveh Razavi, and Ian Kash. R2c2: A network stack for rack-scale computers. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 551–564, 2015.

[14] William J Dally. Virtual-channel flow control. *ACM SIGARCH Computer Architecture News*, 18(2SI):60–68, 1990.

[15] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, February 2013.

[16] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, Seattle, WA, April 2014. USENIX Association.

[17] G. Faanes, A. Bataineh, D. Roweth, T. Court, E. Froese, B. Alverson, T. Johnson, J. Kopnick, M. Higgins, and J. Reinhard. Cray cascade: A scalable hpc system based on a dragonfly network. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–9, 2012.

[18] Andrew Ferguson, Steve Gribble, Chi-Yao Hong, Charles Killian, Waqar Mohsin, Henrik Muehe, Joon Ong, Leon Poutievski, Arjun Singh, Lorenzo Vicisano, Richard Alimi, Shawn Shuoshuo Chen, Mike Conley, Subhasree Mandal, Karthik Nagaraj, Kondapa Naidu Bollineni, Amr Sabaa, Shidong Zhang, Min Zhu, and Amin Vahdat. Orion: Google's software-defined networking control plane. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021.

[19] Peter X. Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. pHost: Distributed Near-optimal Datacenter Transport over Commodity Network Fabric. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15, pages 1:1–1:12, New York, NY, USA, 2015. ACM.

[20] Marina García, Enrique Vallejo, Ramón Beivide, Miguel Odriozola, and Mateo Valero. Efficient routing mechanisms for dragonfly networks. In *2013 42nd International Conference on Parallel Processing*, pages 582–592. IEEE, 2013.

[21] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Evolve or die: High-availability design principles drawn from googles network infrastructure. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 58–72, New York, NY, USA, 2016. Association for Computing Machinery.

[22] R. L. Graham, D. Bureddy, P. Lui, H. Rosenstock, G. Shainer, G. Bloch, D. Goldenerg, M. Dubman, S. Kotchubievsky, V. Koushnir, L. Levi, A. Margolin, T. Ronen, A. Shpiner, O. Wertheim, and E. Zahavi. Scalable hierarchical aggregation protocol (sharp): A hardware architecture for efficient data reduction. In *2016 First International Workshop on Communication Optimizations in HPC (COMHPC)*, pages 1–10, 2016.

[23] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, page 51–62, New York, NY, USA, 2009. Association for Computing Machinery.

[24] The P4.org API Working Group. P4 Runtime Specification. https://p4.org/p4runtime/spec/v1.2.0/P4Runtime-Spec.html, 2020.

[25] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichik, and Marcin Mojcik. Re-architecting Datacenter Networks and Stacks for Low Latency and High Performance. In *Proceedings of the ACM SIGCOMM 2017 Conference*, SIGCOMM '17, pages 29–42, New York, NY, USA, 2017. ACM.

[26] Hong Linh Truong, W. W. Ellington, J. Y. Le Boudec, A. X. Meier, and J. W. Pace. Lan emulation on an atm network. *IEEE Communications Magazine*, 33(5):70–85, 1995.

[27] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined wan. In

Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM, SIGCOMM '13, page 3–14, New York, NY, USA, 2013. Association for Computing Machinery.

[28] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 1–12, New York, NY, USA, 2017. Association for Computing Machinery.

[29] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 113–126, San Jose, CA, April 2012. USENIX Association.

[30] John Kim, Wiliam J Dally, Steve Scott, and Dennis Abts. Technology-driven, highly-scalable dragonfly topology. In *2008 International Symposium on Computer Architecture*, pages 77–88. IEEE, 2008.

[31] John Kim, William J Dally, and Dennis Abts. Flattened butterfly: a cost-efficient topology for high-radix networks. In *Proceedings of the 34th annual international symposium on Computer architecture*, pages 126–137, 2007.

[32] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, et al. Onix: A distributed control platform for large-scale production networks. In *OSDI*, volume 10, pages 1–6, 2010.

[33] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 514–528, New York, NY, USA, 2020. Association for Computing Machinery.

[34] Sergey Legtchenko, Nicholas Chen, Daniel Cletheroe, Antony Rowstron, Hugh Williams, and Xiaohan Zhao. Xfabric: A reconfigurable in-rack network for rack-scale computers. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 15–29, 2016.

[35] Yuliang Li, Gautam Kumar, Hema Hariharan, Hassan Wassel, Peter Hochschild, Dave Platt, Simon Sabato, Minlan Yu, Nandita Dukkipati, Prashant Chandra, and Amin Vahdat. Sundial: Fault-tolerant clock synchronization for datacenters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1171–1186, 2020.

[36] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. F10: A fault-tolerant engineered network. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 399–412, Lombard, IL, April 2013. USENIX Association.

[37] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, and et al. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 399–413, New York, NY, USA, 2019. Association for Computing Machinery.

[38] Nick McKeown. The islip scheduling algorithm for input-queued switches. *IEEE/ACM transactions on networking*, 7(2):188–201, 1999.

[39] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. 38:69–74, 2008.

[40] Michael Mitzenmacher, Andréa W. Richa, and Ramesh Sitaraman. The power of two random choices: A survey of techniques and results. In *Handbook of Randomized Computing*, pages 255–312. Kluwer, 2000.

[41] Jeffrey C Mogul, Drago Goricanec, Martin Pool, Anees Shaikh, Douglas Turk, Bikash Koley, and Xiaoxue Zhao. Experiences with modeling network topologies at multiple levels of abstraction. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 403–418, 2020.

[42] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 221–235, New York, NY, USA, 2018. ACM.

[43] Tim Nelson, Andrew D Ferguson, Michael JG Scheer, and Shriram Krishnamurthi. Tierless programming and reasoning for software-defined networks. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 519–531, 2014.

[44] Open Networking Foundation. Mission of open networking foundation. https://opennetworking.org/mission/, 2021. Accessed: 2021-03-08.

[45] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the Social Network's (Datacenter) Network. In *Proceedings of the ACM SIGCOMM 2015 Conference*, SIGCOMM '15, pages 123–137, New York, NY, USA, 2015. ACM.

[46] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan RK Ports, and Peter Richtárik. Scaling distributed machine learning with in-network aggregation. April 2021.

[47] Leah Shalev, Hani Ayoub, Nafea Bshara, and Erez Sabbag. A cloud-optimized transport protocol for elastic and scalable hpc. *IEEE Micro*, 40(6):67–73, 2020.

[48] Arjun Singh. *Load-balanced routing in interconnection networks*. PhD thesis, Stanford University, 2005.

[49] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Hanying Liu, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *SIGCOMM '15*, 2015.

[50] Arjun Singhvi, Aditya Akella, Maggie Anderson, Rob Cauble, Harshad Deshmukh, Dan Gibson, Milo MK Martin, Amanda Strominger, Thomas F Wenisch, and Amin Vahdat. Cliquemap: productionizing an rma-based distributed caching system. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 93–105, 2021.

[51] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F Wenisch, Monica Wong-Chan, Sean Clark, Milo MK Martin, Moray McLaren, Prashant Chandra, Rob Cauble, et al. 1rma: Re-envisioning remote memory access for multi-tenant datacenters. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 708–721, 2020.

[52] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P Brighten Godfrey. Jellyfish: Networking data centers randomly. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 225–238, 2012.

[53] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[54] Anduo Wang, Xueyuan Mei, Jason Croft, Matthew Caesar, and Brighten Godfrey. Ravel: A database-defined network. In *Proceedings of the Symposium on SDN Research*, pages 1–7, 2016.

[55] Junlan Zhou, Malveeka Tewari, Min Zhu, Abdul Kabbani, Leon Poutievski, Arjun Singh, and Amin Vahdat. WCMP: Weighted Cost Multipathing for Improved Fairness in Data Centers. In *Proceedings of the Ninth European Conference on Computer Systems*, page Article No. 5, 2014.

[56] Noa Zilberman, Gabi Bracha, and Golan Schzukin. Stardust: Divide and conquer in the data center network. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 141–160, 2019.

A   HARDWARE PACKAGING DETAILS

Incremental deployment is a much more significant consideration for datacenter systems than for supercomputers which are typically installed as a single system, or in a number of predefined phases. Incremental network deployment is challenging for the Dragonfly topology, where growing the size of the fabric requires the topology to be reconfigured to fully exploit the available chip bandwidth. To avoid recabling for expansion, which is hard to reconcile with the availability requirements of a datacenter, we developed a packaging strategy that allows all the networking infrastructure to be landed as one initial deployment, with the servers being populated incrementally as required.
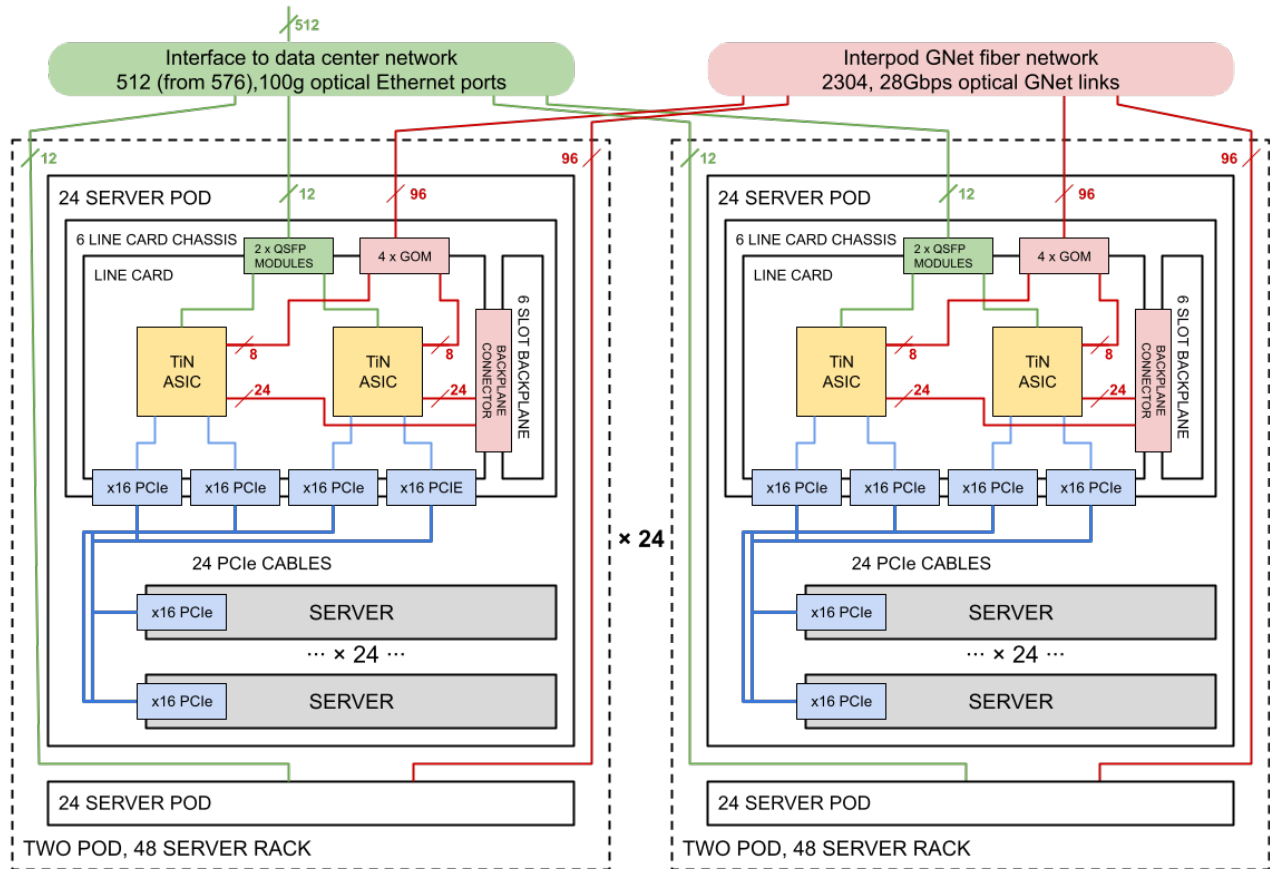
Figure 12: Aquila Clique with 1152 servers, in 24 racks.

A second key consideration was whether to use a blade based system, with a combined server and networking packaging solution, or to work with our existing servers designed around conventional NICs. The latter approach was chosen to avoid having to support two packaging variants of each different server type. These two decisions broadly determined our packaging design.

The physical design (Figure 12) supports up to 48 machines per rack, organized as two pods of 24 servers. The Aquila networking for each pod is provided by a switch chassis containing 12 TiN ASICs, on 6 line cards. The first level interconnect of the Dragonfly is implemented in copper on the switch chassis backplane. Servers are connected to the switch chassis using a cabled x16 Gen 3 PCIe bus. Sideband signals on the cable carry the independent machine management interface from the TiN chip that connects to the server's NC-SI port.

The overall Aquila Clique consists of 24 racks. The connectivity between the racks is optical using custom low cost VCSEL based 4 channel GNet optical modules, 4 per line card. This gives a total of 96 optical GNet connections for the global interconnect level of the Dragonfly from each pod. As there are a total of 48 pods in a clique there are two optical GNet global links between any pod pair. If we connected these directly with two channel fiber ribbons this would re-

quire 47x48/2 = 1128 unique interpod cables to be connected. To simplify the rack to rack cabling we use fiber shuffles within groups of 4 pods to consolidate into wider fiber ribbons allowing the use of 8 GNet link, MPO16 fiber cables. This reduces the rack to rack cabling to 66 4-cable bundles running between 12 pairs of racks greatly simplifying the fiber deployment.

The total number of available 100g Ethernet ports available for connection to the data center spine network from the TiN ASICs is 576. 24 of these are used for rack management. Either 256 or 512 ports are connected to the higher level Ethernet fabric with the remaining 40 ports unused.

A.1 Failure Domains

A key consideration of the Aquila architecture was to reduce the blast radius of any networking component failure. In a conventional network the loss of a TOR impacts all the attached servers; this could be as many as 48 machines for a high radix switch device. In contrast, with the Aquila architecture, loss of a TiN ASIC impacts a maximum of two servers. In practice because the physical packaging solution uses a pair of TiN ASICs on a single line card, the effective blast radius for a repair operation can be up to four servers if 2 servers share a TiN. A switch chassis failure impacts a maximum of 24 servers, however the only chassis components with a significant failure rate are the fans, and N+2 fan redundancy
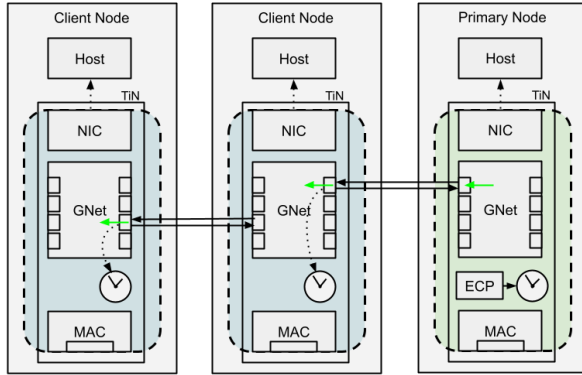
Figure 13: Aquila Clock Sync.

is implemented to minimize the possibility of a chassis level failure.

B   CLOCK SYNCHRONIZATION

B.1   Overview

The timesync protocol on Aquila was designed with the aim of keeping the software overhead for timesync low while also providing a tight bound on the notion of current time across the TiNs in the clique. This protocol maintains a single primary clock in the clique against which clients are synchronized purely in hardware (Figure 13). Synchronization is carried out over the GNet links on the switch side of the TiN by transmitting information as lightweight, 8-Byte "ordered sets" between cells, a class of which (TimeSync) are defined for Clique time synchronization. Clients in the host, expecting an IEEE 1588-like protocol to maintain time in the NIC, are able to query the value of this clock. The hardware also corrects for link delays between neighboring TiNs and for time spent within the chip while waiting for a gap between cells to get on the link.

B.2   Implementation

The Timesync hardware on TiN maintains the current time by counting cycles of the core clock along with status bits which tracks several parameters that indicate the accuracy of

the clock. The value of time is also updated by the reception of timesync messages from the neighboring TiN if the current TiN has been configured to be a client node in the time distribution network. The protocol relies on software to set up this time distribution tree [35].

Once the time distribution tree is configured, the TiN transmits TimeSync ordered sets on a configured number of output GNet links at a fixed interval (typically, about 100us). On a client node, an incoming TimeSync message also causes an update to be sent downstream even if the configured interval between messages has not expired. This is to ensure that even the farthest nodes in the time distribution tree do not drift much from the primary node.

The TimeSync message cannot interrupt a cell on the wire, so the ordered set can wait up to 128ns to get onto the wire. Regardless of the delay, the ordered set indicates the actual time of transmission (+/- 2.5ns) by incrementing the value of current time in the TimeSync message for each cycle that it waits to get onto the wire, including flight time across the chip from the hardware clock, arbitration time to get onto the wire, etc. Each receiving TiN is configured to receive TimeSync messages only on a single port and it adjusts for any on-chip delays to get the TimeSync message to the hardware clock along with the delay through the GNet channel.

The delay through the GNet channel is configured on the receiver by running round trip delay measurement at the time of setting up of the time distribution tree. This is done by the GNet ports by putting them in a "latency measurement" mode where the neighbors exchange special ordered sets and reflect the delay through the channel to software as a status.

On reception of a Timesync message, the client node, checks the validity of the message through comparison of status bits transmitted with the message and the difference between the incoming time and the current time against a configurable threshold. The update to current time is only applied when valid Timesync messages are received and if enough invalid messages are seen, the client node signals that a failure is detected. The protocol relies on software to take action once failure is detected

RDC: Energy-Efficient Data Center Network Congestion Relief with Topological Reconfigurability at the Edge

Weitao Wang[†], Dingming Wu[*], Sushovan Das[†], Afsaneh Rahbar[†], Ang Chen[†], and T. S. Eugene Ng[†]

[†]Rice University, [*]Bytedance Inc.

Abstract

The *rackless data center* (RDC) is a novel network architecture that logically removes the rack boundary of traditional data centers and the inefficiencies that come with it. As modern applications generate more and more inter-rack traffic, the traditional architecture suffers from contention at the core, imbalanced bandwidth utilization across racks, and longer network paths. RDC addresses these limitations by enabling servers to logically move across the rack boundary at runtime. Our design achieves this by inserting circuit switches at the network edge between the ToR switches and the servers, and by reconfiguring the circuits to regroup servers across racks based on the traffic patterns. We have performed extensive evaluations of RDC both in a hardware testbed and packet-level simulations and show that RDC can speed up a 4:1 oversubscribed network by $1.78\times \sim 3.9\times$ for realistic applications and more than $10\times$ in large-scale simulation; furthermore, RDC is up to $2.4\times$ better in performance per watt than a conventional non-blocking network.

1 Introduction

The importance of the data center network (DCN) has led to a series of DCN architecture proposals [26,43,44,53,59,62,66, 74,80,82,84–86,96,110,118] over the past decade. Although these proposals have competing designs for the network core, the designs for the network edge are similar: servers organized in racks. The network core connects multiple racks, and each rack hosts tens of servers that are connected via a Top-of-Rack (ToR) switch. Standardized racks enable unified power supply and cooling, as well as significant space and cable savings. This rack-based topology and connectivity pattern is deeply ingrained in the design of existing DCN architectures.

While traffic within a rack experiences no congestion, traffic across racks often has to contend for bandwidth due to oversubscription in the network core[1]. At the same time, traffic across racks is increasing in data center workloads [36, 37, 40, 99]. Firstly, more and more DCN traffic is escaping the rack boundary due to resource fragmentation [61], large-scale jobs [24], specific application placement constraints for fault tolerance [13], and service-based rack organization for operational convenience [99]—e.g., one rack may host storage servers, and another rack may host cache

servers. Secondly, there is also an increasing amount of traffic that leaves the pod. For instance, a web-frontend cluster may need to retrieve data from a database cluster or submit jobs to a Hadoop cluster [99].

Thus, the need for efficient handling of cross-rack traffic has motivated numerous approaches; but they have one thing in common – they view the rack design (i.e., a ToR switch connecting tens of servers) as a given. Firstly, the non-blocking network and its alternatives [26, 62, 64, 65, 82, 109] aim to enlarge the capacity of the network core. However, due to the scaling limit of CMOS-based electrical packet switches [6, 33, 34, 49, 50, 57, 91, 104, 105], building such a network while staying within the datacenter power budget is challenging [107]. Secondly, rack-level reconfigurable networks [53, 74, 80, 110, 118] add additional bandwidth between the most intensively communicating racks with extra cables, lasers, or antennas to relieve the bottleneck at the core. However, the performance improvement is constrained by the fact that the number of additional paths is usually limited. Thirdly, smarter job placement and execution strategies [39,40,45,46,71,72,87,108,116,120] can also reduce the inter-rack traffic by arranging the jobs based on their traffic pattern. However, these placement solutions cannot perform well if traffic patterns fluctuate at runtime or if the application dictates placement and forces the traffic to be cross-rack.

This paper studies a complementary and little-explored point in the design space, which we call the *rackless data center (RDC)* architecture. It logically removes the fixed, topological rack boundaries while preserving the benefits of rack-based designs, e.g., organized power supply and cooling, and space efficiency. In RDC, servers are still mounted on physical racks, but they are not bound statically to any ToR switch. Rather, they can move logically from one ToR to another. Under the hood, this is achieved by the use of the circuit switches (CS), which can be dynamically reconfigured to form different connectivity patterns. In other words, servers remain immobile, but circuit changes may shift them to different topological locations. Therefore, this new architecture is not committed to any static configuration, so servers that heavily communicate with each other can be grouped on demand, and they can be regrouped as soon as the pattern changes again. Such dynamic server regrouping enabled by RDC leads to performance benefits in many common, real-world scenarios (details in §2).

---

[1]The literature suggests that there exists a wide-range of common oversubscription ratios between 4:1 to 20:1 [43,62,99,105].

We make the following contributions: 1) a novel architecture called RDC, which can be reconfigured to connect servers under different racks in the same logical locality group despite physical rack boundaries; 2) a low-latency RDC control plane and algorithms, which continuously optimize the RDC topology based on the traffic patterns; 3) a prototype of RDC in both testbed and simulation settings, demonstrating that RDC boosts the performance of a 4:1 oversubscribed network by $1.78\times \sim 3.9\times$ for realistic applications and more than $10\times$ in large-scale simulation; furthermore, RDC is up to $2.4\times$ better in performance per watt than a conventional non-blocking network.

2 Motivation

RDC is motivated by inefficiencies that stem from the inherent rack boundaries in today's data centers. RDC enables dynamic topological reconfiguration to regroup servers, leading to improved performance for modern workloads. We propose to realize RDC using circuit switching technologies.

2.1 Rack sizes are inherently limited

Today's DCNs are organized in physical racks as the basic unit. Communication within a rack is through a ToR switch and enjoys lower latency and higher throughput than that across racks. This rack boundary is stressed by a combination of two trends. First, applications are becoming data-intensive. DNN training, iterative machine learning, HPC, big data frameworks (MapReduce, Spark, HDFS) and many other workloads require extensive data communication. Second, the advent of domain-specific accelerators (GPUs, TPUs) and non-volatile memories (NVM) is further shifting the major bottleneck from computation to network IO. The convergence of these trends leads to the need to maximize rack-level performance as much as possible. Broadcom's Tomahawk-4 64x400 Gbps—the fastest Ethernet switch ASIC commercially available on the market today [7]—only supports a rack boundary of tens of servers while maintaining maximum rack-level performance. A few years ago, the End-of-Row architecture was proposed as an alternative, where multiple racks of low port speed servers were connected to a high-radix edge switch to form a larger logical rack [1]. However, high-radix switching is not feasible at high port speeds: 400 Gbps ports are common today, and Ethernet standards are growing to terabit level. Therefore, in the foreseeable future, the physical rack boundaries of tens of servers are here to stay. New solutions are necessary to mitigate inter-rack-level bottlenecks.

2.2 Rack boundaries introduce bottlenecks

1. Jobs fragmented across racks. A job may spread across racks if rack resources are fragmented. This is partly because cluster schedulers assign resources to their own jobs locally [5, 11, 12]; also, dynamic job churns ensure that rack resources aren't always neatly packed [60, 95]. Such resource fragmentation leads to heavy inter-rack traffic which contends for bandwidth due to oversubscription.

2. Workloads with dynamic traffic patterns. Many data-intensive applications (e.g., DNN training, HPC) consist of multiple stages, and each stage has a different yet predictable traffic pattern. For example, Distributed Matrix Multiplication (DMM) has broadcast (one-to-many) and shift (one-to-one) traffic patterns among different subsets of servers in every iteration (Fig. 8(e)). When these jobs coexist in a cluster, the overall combined traffic pattern will change dynamically and predictably. For such workloads, no static job allocation is sufficient to localize all the traffic patterns simultaneously.

3. Applications with placement constraints. Applications may intentionally spread their instances across racks to balance load [55] to reduce synchronized power consumption spikes [70], or to achieve fault tolerance [13]. For instance, to increase resilience, some distributed storage systems, like HDFS, require at least one replica to be placed on a different rack. These requirements result in placement constraints that are by design crossing rack boundaries.

4. Imbalanced out-of-pod traffic. In large datacenters, traffic patterns across racks are often skewed, and out-of-pod traffic demand for each rack is different. For example, only 7.3% of the traffic from the frontend servers is inter-pod, comparing to 40.7% for the cache servers [41, 99]. Operationally, data centers tend to group servers based on their types [99]. So, the above heterogeneity of the out-of-pod traffic demand will make some racks' uplinks highly congested (e.g., cache) while other racks still have unused bandwidth (e.g., frontend).

2.3 Facebook trace analysis: A case study

Methodology. We used a public dataset released by Facebook, which contains packet-level traces collected from their production data centers in a one-day period. The traces were collected from the "frontend", "database", and "Hadoop" clusters, sampled at a rate of 1:30 k, and each packet contains information about the source and destination servers [4]. To understand the benefits of removing rack boundaries, we simulate a rackless design by regrouping servers of different racks into "logical" racks using the algorithms presented in §3. We have two major findings.

Observation #1: Intensive inter-rack traffic. The first observation from the traces is that most of the traffic crosses rack boundaries in a pod. Fig. 1(a) shows the heatmap of traffic pattern inside a frontend pod with 74 racks, collected during a 2-minute interval. If a server in rack i sends more traffic to another server in rack j, then the pixel (i, j) in the heatmap will become darker. Intra-rack traffic appears on the diagonal (i.e., $i = j$). The scattered dots show that the traffic does not exhibit rack locality—in fact, 96.26% of the traffic in this heatmap is inter-rack but intra-pod. A similar trend exists for the database trace: 92.89% of traffic is inter-rack but intra-pod. Hadoop trace has more intra-rack traffic but still has 52.49% of traffic being inter-rack but intra-pod.

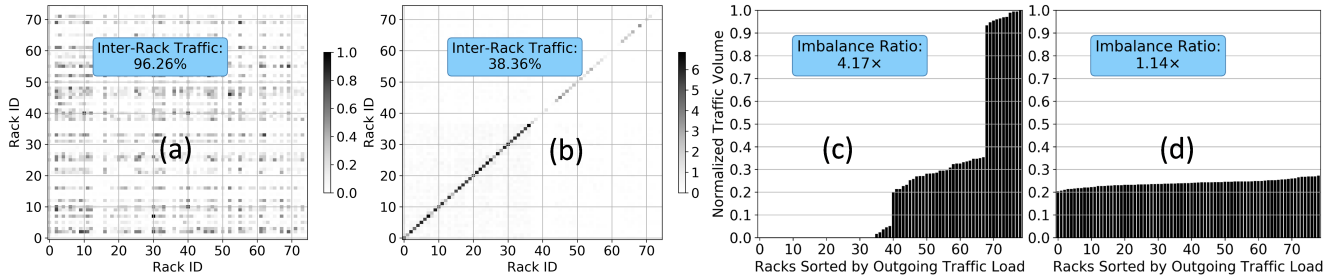**Implication #1: Regrouping servers improves locality.** Fig.

Figure 1: Traffic patterns from the Facebook traces. (a) is the rack-level traffic heatmap of a representative frontend pod. (b) shows the heatmap after regrouping servers in (a). (c) and (d) plot the sorted load of inter-pod traffic across racks in a representative database pod, before and after server regrouping, respectively.

1(b) shows the heatmap if servers are regrouped under different racks based on their communication intensity, simulating the desired effect of RDC. Here, most of the traffic is on the diagonal, and inter-rack traffic is reduced significantly to 38.4%. Assuming a 4:1 oversubscribed network, what used to be inter-rack traffic now enjoys 2.82x higher bandwidth. For the database and Hadoop traces, the inter-rack traffic ratios after regrouping are 28.4% and 41.6%, respectively.

Observation #2: Out-of-pod traffic imbalance. Another notable trend is the heavy imbalance of out-of-pod traffic. Fig. 1(c) sorts the racks based on the amount of out-of-pod traffic they sent (traffic trace: database) in a 20-min interval, where the X-axis is the rack ID, and the Y-axis is the (normalized) out-of-pod traffic volume. As we can see, the top 11 racks account for nearly 50% of the out-of-pod traffic, and almost half of the racks never sent traffic across pods. Therefore, some uplinks of ToR switches are heavily utilized, whereas other links are almost always idle. The load imbalance, defined as $max(L_i)/avg(L_i)$, where L_i is the amount of out-of-pod traffic from rack i, is as much as 4.17. We found qualitatively similar results on other traces.

Implication #2: Grouping servers mitigates load imbalance. Fig. 1(d) shows the results if servers can be regrouped. In the simulated RDC network, the inter-pod traffic is much more evenly load-balanced across racks, achieving a load imbalance of 1.14. Moreover, the aggregated bandwidth for the out-of-pod traffic increases to 1.79x of the previous bandwidth. This would make better use of the ToR uplinks and avoid congesting any particular link due to imbalance.

2.4 The Power of RDC

Driven by the application-level demand and trace-based analysis, we propose the concept of *rackless data center (RDC)*, which logically removes the physical rack boundaries while maintaining the high-speed rack-level performance. In RDC, servers are mounted on the same "physical rack" sharing the power supply and cooling system but can be logically moved across the ToR switches. We call the new groups of servers served by the same ToR a "logical rack". Fig. 2 illustrates the benefits of RDC due to server regrouping.

1. Mitigate the effect of resource fragmentation. RDC can
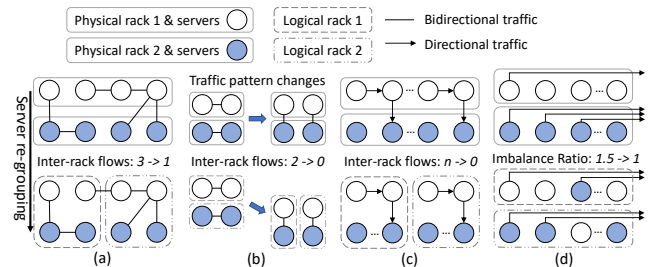
Figure 2: Comparisons between before and after server regrouping for (a) placement optimization, (b) dynamic optimization for evolving patterns, (c) application constraints accommodation, and (d) out-of-pod load balancing.

reduce the effect of resource fragmentation by relocating the heavily communicating server groups under the same logical rack, thus reducing inter-rack traffic. RDC can completely localize smaller jobs that are possible to be packed within one logical rack, like the job on the left-hand side of Fig. 2(a). Even for bigger jobs that cannot be packed within one logical rack, RDC benefits them by (1) localizing as many traffic flows as possible to logical racks, like the job on the right-hand side of Fig. 2(a); and (2) minimizing overall inter-rack traffic from all jobs, leaving the core bandwidth to be shared by much fewer flows that must cross the rack boundaries.

2. Optimize for dynamic traffic patterns. The ability of dynamic server regrouping enabled by RDC can potentially optimize the applications with variable yet predictable traffic patterns. With such changing patterns as shown in Fig. 2(b), RDC is able to dynamically change the topology and minimize the inter-rack traffic for all patterns.

3. Accommodate application placement constraints. As shown in Fig. 2(c), application-level constraints can be accommodated by RDC while localizing traffic. For example, HDFS always requires at least one data block replica to be placed on a different rack. By regrouping the servers from different racks into one logical rack, RDC can place the replicas to a different physical rack but within the same "logical" rack, which provides higher bandwidth and also satisfies the replica placement policy of HDFS.

4. Balance out-of-pod traffic. RDC is able to regroup the servers according to their out-of-pod traffic demands and balance link utilization, hence relieving the bottleneck. In
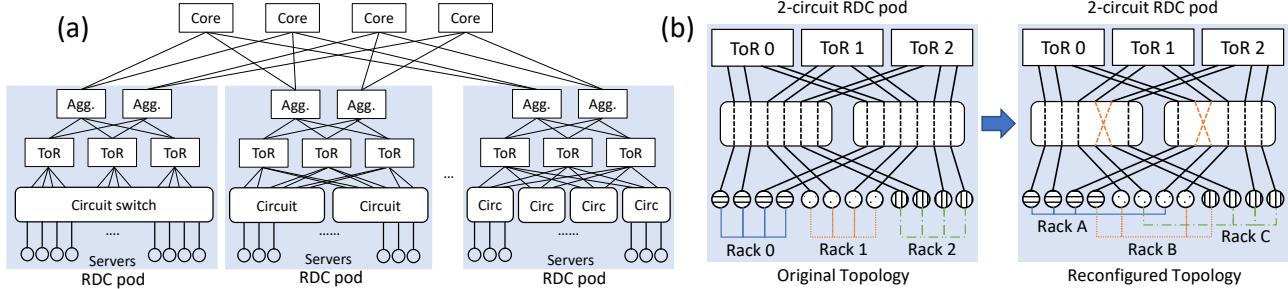
Figure 3: RDC architecture overview. (a) is an example of the RDC network topology. Different numbers of circuit switches can be inserted at the edge between servers and ToR switches. Connectivities for aggregation switches (*agg.*) and core switches remain the same as in traditional Clos networks. (b) shows the original topology and an example reconfigured topology for a 2-CS RDC pod with 3 racks and 4 servers under each rack.

Fig. 2(d), the imbalance ratio has been decreased to 1 from 1.5 after the grouping is changed according to the out-of-pod traffic demand.

2.5 Realizing RDC

Circuit switches (CS) are widely used to provide reconfigurable connections among end points, which is a great fit for the server regrouping functionality of RDC that we discussed above. One realization of RDC is to connect all the servers and all the ToR switches within a pod with a single CS. Alternatively, RDC can also use multiple smaller port count CSes to form a distributed reconfigurable server-to-ToR fabric.

RDC can potentially leverage any kind of CS technologies, including optical and electrical circuit switches alike [104]. However, at high data rates, optical transceivers are the standard interfaces. Therefore, to make the realization long-term sustainable, we consider various optical circuit switching (OCS) technologies. Several OCS technologies are available today such as 3D/2D MEMS, AWGR, etc. Fundamentally, OCS does not perform packet-level processing and forwards the photon beams using mirror rotation, diffraction, etc., which leads to some inherent advantages such as a) agnostic to data-rate (or modulation format), b) negligible power consumption, c) negligible forwarding latency due to no buffering, and d) no need of transceivers at the OCS ports. Additionally, different OCS technologies can provide very fast switching. For example, 2D-MEMS-based OCSes provide microsecond switching [96]), AWGR switches with the latest tunable transceivers can provide nanosecond switching [33, 35, 48, 49, 58, 77]. Moreover, OCS are highly reliable [101] and, due to their simplicity, mostly free from firmware bugs and software misconfigurations.

3 The RDC Architecture

3.1 Connectivity structure

RDC changes the traditional multi-layer Clos topology [26, 62] by inserting one or more circuit switches (CS) at the edge layer between the servers and ToR switches, so that the server can be connected to different ToR switches through circuit reconfiguration. The aggregation and core layers of the net-

work remain the same. Each circuit switch has some ports connected to every ToR switch within the pod to guarantee that the servers could be connected to any ToR switch. For the 1-CS RDC pod, the servers can be grouped without constraints, as long as the number of servers under each ToR switch is the same. If multiple circuit switches are used in one pod, the additional connectivity constraint is that not all the servers under one circuit switch can be connected to the same ToR. With such design, RDC maximizes the flexibility to permute the server-ToR connectivities, allowing the most intensively communicating servers to be localized under the same ToR and enjoy the line rate throughput.

Fig. 3(a) shows an example of the RDC pods. For a pod with *m* racks and *n* servers per rack, 2*mn* ports should be provided by all the circuit switches in total to link both servers and ToR switches. For instance, a 16-rack pod with 32 servers can be built with either 1 circuit switch with 1024 ports or *k* switches with $\frac{1024}{k}$ ports each. Fig. 3(b) gives a detailed example of inserting multiple circuit switches and how to reconfigure for regrouping servers. For a pod with *k* circuit switches, $\frac{n}{k}$ servers under each ToR are connected to one circuit switch, so that the original topology can keep every server under its own physical ToR switch.

Intuitively, if we increase the number of CSes, the design becomes more distributed which decouples it from a particular CS technology's port count availability; while at the same time, the flexibility of moving servers across the ToRs is slightly reduced. To shed light on this trade-off, we perform trace-based analysis with varying numbers of CSes between the servers and ToR switches. To find a valid server regrouping, we formulate an Integer Linear Programming (ILP) which maximizes the traffic localization given the constraints arising from multiple CSes (more details in §4.3). For the analysis, we consider an RDC pod with 16 ToRs and 32 servers per ToR, having 4 : 1 oversubscription above the ToR level. We vary the number of CS from 1 to 8 and compare the performance with a static 4 : 1 oversubscribed network. Fig. 4 shows a boxplot of the flow completion times (FCT) of these architectures for flow-level Cache traffic trace generated from [99]. We observe that the potential benefit of RDC remains high
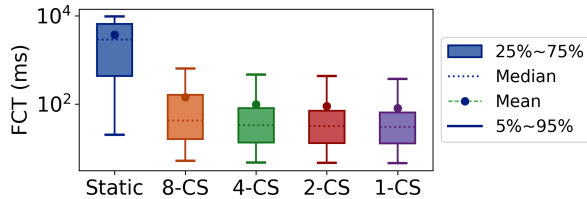
Figure 4: The potential improvement for FCT remains high across a wide range of multi-CS configurations in RDC

across a wide range of CS configurations, which validates the efficacy of our distributed design.

3.2 The RDC Controller

Today's data centers are constructed from modular pods [3, 14, 19, 22], where a pod typically hosts one type of service. RDC similarly views pods as basic units and uses a per-pod network controller that manages both packet switches and circuit switches within the pod. The controller reconfigures the network at timescales of seconds or longer depending on the traffic pattern. It has two operation modes: it can receive the traffic demands or commands from the applications directly in the proactive mode, or passively monitor the traffic statistics from packet switches in the reactive mode.

We illustrate the workflow for both modes in Fig. 5. The controller 1) first collects the traffic statistics by querying the flow counters on the ToRs, or passively receives the information from the applications; 2) determines the optimized topology with certain optimization goals (§4); 3) generates a set of new routes and pre-installs them on the packet switches; and 4) finally sends the circuit reconfiguration request to the circuit switch and simultaneously activates the new routing rules on packet switches. The first two steps serve as the RDC control plane (discussed later in §4), while the last two steps configure the data plane (discussed in §3.3). Note that only the final step would cause a small amount of disturbance due to the circuit reconfiguration delay.

3.3 Routing

In traditional DCNs, forwarding rules are aggregated based on IP prefixes. In RDC, such aggregation does not work as servers have no fixed locations. Instead, RDC uses per-pod flat IP addressing and exact matching rules on packet switches. Topology changes are captured by updating the routing rules. These rule updates are for intra-pod routing only, as routing mechanisms across pods remain unchanged.

In an RDC pod, each ToR has a flow table entry for every server IP in its rack, and a single default entry for other addresses outside the rack. Each ToR splits traffic to other racks equally across its uplinks using ECMP [69]. All *agg.* switches have the same forwarding table: one entry per destination IP. The flow entries on ToRs and *agg.* switches both need to be updated when topology changes. For ToRs, only the rules for downward traffic need to change; the default ECMP entry for upward traffic remains the same. Therefore, for an RDC pod
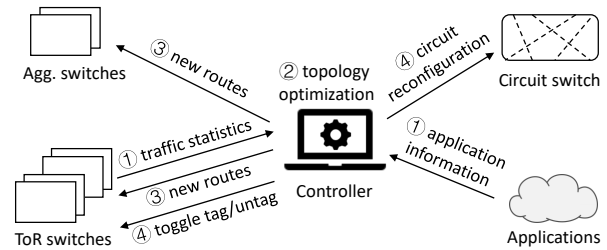
Figure 5: Workflow overview of RDC.

with m racks and n servers per rack, a topology change could result in n rule updates on ToRs and $m \times n$ updates on *agg.* switches, which is on the order of hundreds to a thousand. Updating this number of rules on an OpenFlow switch could take 100ms to over 1s [68, 76]. Previous works have developed the *two-phase commit* method to reduce disruption during updates [81, 98], which first populate the switches with new routing rules and then flip the packet version at the ingress switches. However, such an approach cannot avoid packet loss in the transient state, like changing the packet version rule [81]. This is because updating the packet version rule at the ingress switch requires two rule changes—removing the old rule and installing the new rule—and therefore is not atomic. Our measurement on a Quanta T3048-LY2R Open-Flow switch shows the transient period could last for 0.5ms.

Instead of changing the packet version, in RDC a switch performs binary changes from VLAN tagging packets to not tagging them, and vice versa. The VLAN-tagged packets will match a group of rules with the VLAN tag as a match field, whereas the packets without VLAN tags will match a more general group of rules without VLAN IDs. In this way, adding and removing a single VLAN tag rule achieves the same goal as changing the packet version, but the operation is atomic and avoids packet loss (more details in §A.1.) We apply this update approach to both ToR and *agg.* switches but only tag or not tag packets on ToR switches. Tag flipping actions are only performed when the new forwarding rules have been populated network-wide. The VLAN tag flipping actions need to be executed at the same time across multiple ToRs; such network-wide changes can be performed using well-known SDN time synchronization [90] and consistent update [42, 73, 100] techniques, so that changes can be synchronized and take effect atomically.

3.4 Discussions

Reducing the path length: Besides the throughput benefits mentioned in §2, localizing the hosts under the same logical rack would effectively reduce the average path length (evaluated in §5.2), and thus reduce the network latency. Therefore, low-latency applications and disaggregated systems [56, 63, 92, 94, 103] may benefit from RDC's design as well.

ToR failure handling: In a traditional data center, a ToR failure disconnects all servers in the rack. ToR failures are handled either by multi-homing servers to several other ToRs

[79, 83] or by replicating applications under multiple ToRs [113, 119]. But in RDC, servers are not tied to any particular ToR, servers under a failed ToR can be migrated to a healthy ToR. To host the relocated servers, we can reserve some "free" ports on each ToR for recovery or install a set of backup ToRs [112, 114]. Specifically, for an RDC pod with m racks and n servers per rack, we only need $\frac{n}{m}$ free ports per ToR (or n ports overall) to recover from any single ToR failure, which incurs a low additional cost and complexity.

CS failure handling: In general, circuit switches are extremely reliable [102]. Commercial OCS products have more than 28.5 years of mean-time-between-failure (MTBF) and come with redundant control processors [2]. However, if a CS failure happens, only a small fraction of servers will be disconnected uniformly under each ToR of RDC, due to its connectivity structure. The most common failure mode for the CS is the power outage. To mitigate this, multiple redundant power supplies can be used for the CS [2]. For further protection, battery backups can be used—since the CS draws only tens of Watts, a battery backup already goes a long way.

4 RDC Control Algorithms

RDC has a general framework to support various topology optimization algorithms, working in two modes to collect the traffic demand matrix and compute the reconfiguration plan.

4.1 Proactive-mode RDC

The *proactive* mode of RDC allows applications to explicitly call the RDC controller via RPC with two APIs: **1) Traffic demand matrix** can be reported by the applications to request reconfigurations. Along with the demand matrix, RDC controller will request the application to specify one topology optimization algorithm from the algorithms described in §4.3 as well. After receiving the request, the RDC controller will calculate an optimal topology with a specified algorithm and conduct the reconfiguration accordingly. **2) Raw configuration commands** can also be given directly from the applications. For this method, formatted data to describe the new circuit connections will be sent to the controller, so that the controller could bypass the calculation of the optimal topology and directly used the received configuration to initiate the reconfiguration. An additional benefit for applications to send raw configuration plans is that it enables network-aware job placement and scheduling since the applications know the future network requirements in advance.

There are several scenarios where applications can benefit from telegraphing their intent to the RDC controller: 1) In a case where applications intentionally spread their deployment across racks—e.g., for fault tolerance [40] or for reducing synchronized power consumption spikes [70]—inter-rack traffic patterns are unavoidable in traditional architectures. In RDC, however, such applications can request relevant servers to be grouped together logically. 2) The cluster applications may be allocated with resources from multiple racks due to frag-

mentation. By aggregating those fragmented resources to the same logical rack, RDC improves the bandwidth and reduces the average latency. 3) When applications have changing traffic patterns (e.g., distributed matrix multiplication (DMM) algorithms proceed in iterations with shifting traffic patterns), they can request reconfigurations before the next phase starts to ensure locality throughout the job. 4) Last but not least, RDC could rely on the out-of-pod traffic demands reported by applications to balance the load across different ToR uplinks.

We evaluate three different applications in §5.1 to show the performance of proactive-mode RDC, including HDFS, Memcached, and DMM.

4.2 Reactive-mode RDC

The *reactive* mode of RDC does not require to modify application; it collects traffic statistics from the network in one epoch, and reconfigures the network with an optimized topology for the next, based on the statistics and one of the optimization algorithms from §4.3, specified by the network operators.

Traffic statistics. The RDC controller pulls flow counters from ToRs periodically. A flow counter associates the 5-tuple (13 bytes) of a flow to an 8-byte counter value and thus has 21 bytes in total. Switch memory constraint is traditionally the main concern of maintaining per-flow counters, but this constraint is loosening over the years as the switch SRAM size has been continuously growing. The most recent switch ASICs have 50-100MB of SRAM and can store millions of flow states [18, 88]. As recent DCN measurement works show that the number of concurrent flows per server is on the order of hundreds to a thousand [28, 99], each ToR in RDC would then need tens of thousands of flow counters assuming tens of servers per rack. For instance, assuming an RDC pod with 16 racks and 32 servers per rack, and a counter pulling period of 10s, the control channel bandwidth usage is roughly 8.6Mbps, which is low enough to be feasible.

Demand estimation algorithm. Previous works have shown that data center workloads demonstrate certain degrees of stability [38, 99], and RDC similarly relies on this stability to estimate the traffic demand based on historical data. But the *observed* traffic volumes on ToR switches are biased by the *current* topology, so it is important to estimate the true traffic demand, i.e., the traffic demand when flows are not bottlenecked by the network core. Mitigating such observation bias has been studied in previous work, Hedera [27], and we adopt a similar heuristic.

A flow could be bottlenecked either by the network or by the application itself. We call the first class of flows *elastic* and the second *non-elastic*, and RDC only considers elastic flows. The heuristic is to remove flows from the observed traffic matrix whose sizes are smaller than their fair share. The remaining flows are treated as elastic, and RDC calibrates for potential bias in the counters by computing their idealized bandwidth share (i.e., their bandwidth share if they are only bottlenecked by the host NICs' capacity) as the es-

dst \ src	0	1	2	3
0		?	?	?
1	-		?	?
2	?	?		-
3	?	?	-	

dst \ src	0	1	2	3
0		1/3	1/3	1/3
1	-		1/2	1/2
2	1/2	1/2		-
3	1/2	1/2	-	

dst \ src	0	1	2	3
0		1/3	1/3	1/3
1	-		1/2	1/2
2	1/2	1/3		-
3	1/2	1/3	-	

Source-side fair share → Destination-side check →

Figure 6: Hedera demand estimation example. Each "?" represents one flow from source host to destination, "-" represents no flow between that source-destination pair, and number "1/2" represents 50% of host bandwidth. This example ends in one iteration, but it takes more iterations for a more complicated traffic matrix.

timated demand [27]. Hedera is an algorithm to calculate the max-min fair share rate of each flow within a network. It performs multiple iterations to firstly increase the flow capacities at the source (no greater than the source host capacity) and then decrease the exceeding capacities (sum of enlarged flow capacities subtracting the actual NIC capacity) on each destination host until the flows' capacities converge. A simple demand estimation example that ends with only one iteration is shown in Fig. 6 (More details in §A.3). After convergence, the estimated flow demands are aggregated into a server-to-server traffic matrix for reconfiguration. The effectiveness of this demand estimation algorithm is evaluated in §5.4.

4.3 Topology optimization algorithms

RDC enables a range of topology optimization and reconfiguration algorithms.

1. Traffic localization algorithm reconfigures the network to localize inter-rack traffic, after obtaining the flow demands proactively or reactively. The objective of the localization algorithm is to minimize the traffic demands across the logical racks of the new topology. With this objective, the localization algorithm can be formulated as an Integer Linear Programming (ILP) problem as described in §A.4. However, finding the optimal solution is NP-hard, so we provide heuristic alternatives with balanced graph partition [75] for 1-CS RDC and a simplified algorithm for multi-CS RDC discussed in §A.4. The heuristic algorithms can find a high-quality regrouping plan within tens of milliseconds as shown in Table 2.

2. Uplink load-balancing algorithm spreads out-of-pod traffic across ToR switches for load balancing, relieving the potential congestion on the over-subscribed uplinks. The objective for uplink load balancing (ULB) is to minimize the maximum out-of-pod traffic from one rack. We provide a formal problem formulation and faster heuristic algorithms in §A.2.

3. Mixed optimizations can be developed in RDC to localize the inter-rack traffic and balance the out-of-pod traffic at the same time, e.g., for a *mix* of workloads or applications. To satisfy this goal, the objective of this problem will be minimizing $\alpha T + \beta R$, where T is the total inter-rack traffic demands within a pod, R is the maximum volume of out-of-pod traffic across ToRs, and α and β are the respective weights [40].
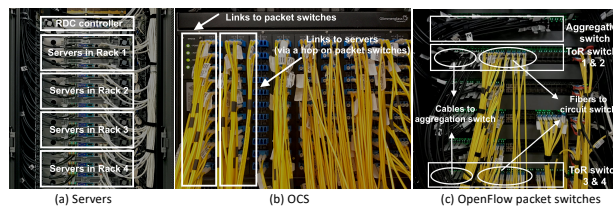
Figure 7: RDC prototype with 4 racks and 16 servers.

4. Scenario-specific optimizations allow applications or network operators to define their own optimization algorithms for regrouping the servers into logical racks. The applications are able to define their own objective function and add more application-specific constraints.

5 Implementation and Evaluation

We conduct comprehensive evaluations using testbed experiments and packet-level simulations. Our experiments focus on several dimensions: a) real-world applications of RDC to HDFS [10], Memcached [23], and MPI-based distributed matrix multiplication (DMM) [54] as use cases, b) packet-level simulations on the latency and throughput improvements at scale, c) packaging, power, and capital cost analysis, and d) microbenchmarks on RDC, including non-disruptive control loop latency.

Testbed. Our RDC prototype consists of 16 servers and 4 ToR switches in 4 logical racks, one *agg.* switch and one circuit switch; Fig. 7 illustrates our hardware testbed. The ToR switches are emulated on two 48-port Quanta T3048-LY2R switches. Each ToR switch has four downlinks connected to the servers, and one uplink to the *agg.* switch, forming an oversubscription ratio of 4:1. We can tune this ratio to emulate a non-blocking network by increasing the number of uplinks to 4. The *agg.* switch is a separate OpenFlow switch. The OCS is a 192-port Glimmerglass 3D-MEMS switch with a switching delay of 8.5 ms. This can also be replaced with other types of OCS. Each server has six 3.5 GHz dual-hyperthreaded CPU cores and 128 GB RAM, running TCP CUBIC on Linux 3.16.5. Most of our experimental results except the large-scale simulation in §5.2 are obtained on this testbed.

Packet-level simulator. In order to simulate a wider variety of experimental settings, we have developed a packet-level simulator based on *htsim*, which was used to evaluate MPTCP [97] and NDP [67]. This simulator has a full implementation of TCP flow control and congestion control algorithms and supports ECMP. We simulate a conservative circuit reconfiguration delay of 8.5 ms, which is what our testbed 3D-MEMS switch achieves. As discussed in §2.5, much faster circuit switching technologies exist [33, 48, 58, 86] that can further improve the performance of RDC. Note that only the circuit that is being reconfigured will experience a disruption; all other circuits continue to function. Packets in flight during reconfiguration will be dropped if they traverse the disrupted links, and unsent packets will be buffered at the servers. We simulate an RDC pod with 512 servers, 32 servers per rack,
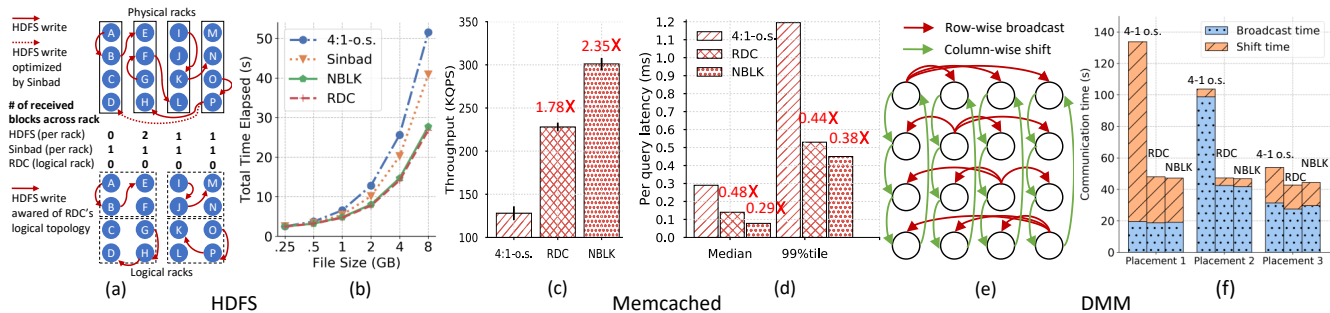
Figure 8: Application performance improvements of RDC compared with the 4:1 oversubscribed network (4:1-o.s.), 4:1-o.s. network powered with Sinbad [45], or the non-blocking network (NBLK). (a) The HDFS write traffic pattern and the number of received blocks per rack. (b) The HDFS transfer time. (c)-(d) Memcached query throughput and latency. (e) The DMM traffic pattern. (f) Average shift time and broadcast time.

and 16 racks overall. The 16 ToR switches are connected to a single *agg.* switch with tunable oversubscription ratios. Results in §5.2 are obtained via simulation.

5.1 Real-world applications

First, we show how RDC can improve the performance of real-world applications for each of its use cases.

HDFS. We set up an HDFS cluster with 16 *datanodes* across 4 racks and 1 *namenode*, with a replication factor of 3 and a block size of 256 MB. All data blocks are cached in the RAM disk to prevent the hard drive from being the bottleneck. The 16 clients initiated concurrent write requests to 16 HDFS files, respectively. According to the default HDFS data block placement policy, when writing a data block to a datanode, a replica of the block will be placed on the same rack of the original copy, and another replica is placed on a remote rack for resilience (Fig. 8(a)). Therefore, a write operation generates an intra-rack flow and an inter-rack flow.

HDFS can localize all the inter-rack traffic (for storing replicas) by using both proactive RDC and network-aware replica placements. Fig. 8(b) shows the performance gain with RDC and compares it with the non-blocking network (NBLK) and an advanced bandwidth-centric replica placement solution, Sinbad [45]). Sinbad keeps track of the paths and links to reach the replicas within the most recent period and assigns the next replica to the least-utilized paths in the recent period. Therefore, Sinbad does not reduce cross-rack traffic, but can relieve bottlenecks at network links by load balancing as shown in Fig. 8(a). Specifically, it detects traffic imbalance for transferring inter-rack replicas and aims to utilize all links roughly equally—i.e., each rack hosts one replica. In the results, we can see that Sinbad improves the total time for HDFS writes, but still underperforms the NBLK network. In contrast, RDC allows the HDFS to regroup servers directly. Moreover, with the new topology, HDFS could change the replica placement scheme to keep all traffic within the logical racks but satisfy fault-tolerance constraints at the same time, as shown in Fig. 8(a). HDFS with RDC achieves similar performance as the NBLK network, reducing the total time to 0.59× on average, compared to the original topology and
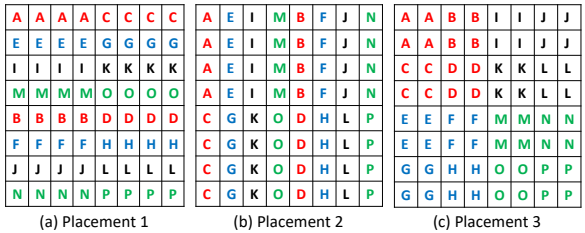
Figure 9: Three different placements for DMM. A-P represent 16 servers and A-D, E-H, I-L, M-P belong to four physical racks separately.

placement policy.

Memcached. We then configured Memcached [23] servers on two racks, and issued read/write requests from two other racks. This emulates the scenario where clients in one pod access cache servers in another pod. Our workload has a) 200 k key-value pairs uniformly distributed across 8 servers, b) a 99%/1% read/write ratio, and c) 512 byte keys and 10 KB values. We adopted a Zipfian query key distribution of skewness 0.99 similar to previous works [31, 93], which led to a load imbalance ratio of ~1.8 on the server racks.

By reallocating the servers with hot keys equally onto every ToR, RDC improves the query throughput by 1.78× on average and reduces the median latency to 0.48× as shown in Fig. 8(c)-(d). These improvements are close to what a non-blocking network could achieve. RDC also cuts the tail latency significantly, for which network congestion is a major cause [30, 117]. Since in the baseline setting, ToR uplink can easily get congested when several hot keys are coincidentally located in the same rack, even if the overall uplink utilization is low. In contrast, RDC can observe the traffic patterns due to the hot keys, and spread the servers hosting these keys to different racks. This reduces the peak uplink utilization.

OpenMPI DMM. We set up a 16-node OpenMPI cluster across 4 racks and implemented a commonly used DMM algorithm [54] with 64 processes. Matrices are divided into 64 blocks (submatrices). Each server has 4 processes to form an 8×8 process layout. Then in each iteration, it performs a "broadcast-shift-multiply" cycle where a process a) broadcasts submatrix row-wise, b) shifts submatrices column-wise, and
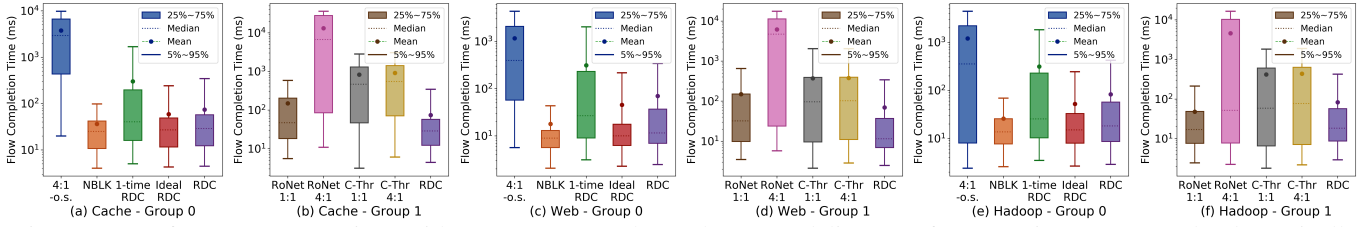
Figure 10: Performance comparison with RDC. Group 0 shows that RDC delivers performance improvements by dynamically reconfiguring the network and achieves similar performance as NBLK; group 1 shows that RDC outperforms alternative designs by benefiting a higher amount of inter-rack traffic.

c) multiplies submatrices as shown in Fig. 8(e). We consider three placements for processes: 1) Fig. 9(a): places them row-wise (no cross-rack traffic for broadcast), 2) Fig. 9(b): places them column-wise (no cross-rack traffic for shift) and 3) Fig. 9(c): places them in a mixed manner, considering both broadcast and shift traffic across racks.

By dynamically configuring the topology for different phases during DMM, RDC shrinks the communication time as well as the end-to-end execution time. Fig. 8(f) shows that RDC improves the overall communication time for placements 1, 2, and 3 by $3.9\times$, $2.3\times$, and $1.26\times$ respectively compared to a static 4:1 oversubscribed network, achieving almost the same performance with the NBLK network. Since the applications have evolving traffic patterns, no static process placement is consistently optimal. Out of the three placements, placement 3 jointly minimizes the cross rack traffic for both communication patterns in DMM, outperforming the other two strategies.

5.2 Performance at scale

Next, we evaluate the reactive RDC pods at the data center scale using the packet-level simulator. Our baselines are a) a static non-blocking network (NBLK), b) a static network with 4:1 oversubscription (4:1-o.s.), c) RDC with future traffic-demand information (Ideal RDC), d) a 4:1-o.s. network that applies RDC's reconfiguration algorithm only once over the entire traffic trace (One-time RDC). e) a hybrid network—like C-Through [110] with 16 4:1/1:1 oversubscribed reconfigurable circuit ToR-pair links in addition to a 4:1-o.s. network, which is similar to Firefly [66], and ProjecToR [59] in terms of performance. f) a novel circuit-core network—RotorNet [86] with 4:1/1:1 oversubscribed ToR uplink bandwidth. Note C-through has the same circuit switching delay as RDC and buffers packets at ToRs during the circuit downtime.

We used the Cache, Web, and Hadoop traffic traces from Facebook. Since the original traces do not contain flow-level information, we generated flow-level traffic based on the sampled packet traces from [99]. Specifically, we inferred the source/destination servers of the flows from the trace, and simulated flow sizes and arrival times based on Figures 6 and 14 in the same Facebook paper. The Cache workload has an average flow size of 680 KB, with 87% being inter-rack. The Web workload has an average size of 63 KB with 96% inter-rack. For the Hadoop workload, the average size is 67.18 KB
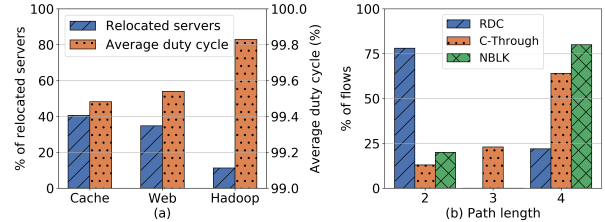
Figure 11: RDC's average circuit duty cycle is >99% even with frequent reconfigurations; RDC has an average path length 35% shorter than NBLK.

but only 60% is inter-rack traffic. All traffic traces last for 30s in the simulation, and RDC's reconfiguration period is 1s.

Fig. 10 shows the boxplot of flow completion times (FCT) for RDC and the baselines using the three traces. We observe that RDC reduces the median FCT by more than an order of magnitude compared to 4:1-o.s. network. Applying RDC's traffic localization algorithm once can bring some improvements on the median FCT but not as significant as RDC and NBLK, since the traffic pattern changes during the simulation. We found that one root cause for the performance improvements is due to TCP dynamics—severe inter-rack congestion causes consecutive packet losses and TCP becomes very conservative in increasing its sending rate. More importantly, we observe that RDC with future knowledge of traffic demands performs consistently close to the non-blocking network, which again demonstrates the power of a rackless network. Without future knowledge, RDC can still achieve similar performance as NBLK with a slightly longer median FCT, because the cache workload is largely stable at the time scale of seconds, similar to that in the Database workload in the original traces. As for other solutions, C-Through's average FCTs are at least $3.21\times$ higher than RDC. Because although C-Through adds extra inter-rack bandwidth, it is provisioned for only 16 ToR pairs. As the traffic traces that motivate our RDC design have more than 16 intensively-communicating ToR pairs (see the heatmap in Fig: 1), C-Through falls short in relieving inter-rack congestion even after enlarging the bandwidths of 16 extra links. 4:1-o.s. RotorNet has the same total uplink bandwidth as RDC, but its performance is much worse than RDC. The non-blocking RotorNet is $2\times$ and $2.17\times$ slower than RDC on the Cache and Web traces; only for the Hadoop traces, it can reduce RDC's average FCT to $0.576\times$. Since RotorNet provides a dedicated
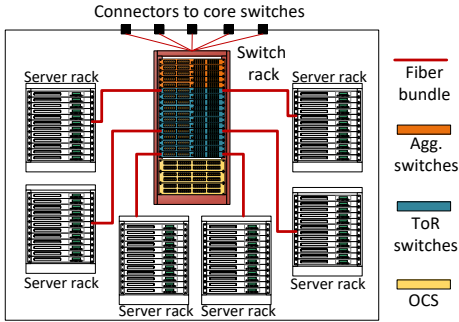
Figure 12: Packaging design of an RDC pod.

link between each ToR pair, if the traffic is skewed between some ToR pairs, it cannot achieve the best performance. So only the Hadoop trace, which has only 60% inter-rack and is quite evenly distributed across different ToRs, enjoys better performance on non-blocking RotorNet.

Fig. 11(a) shows that the average number of servers being relocated in each epoch is different across traces. The duty cycle is an important metric in optical networks to represent the percentage of time that an optical link is up and available for transmission. Assuming one reconfiguration per second, the lowest circuit duty cycle of RDC is 99.2% in theory (details about downtime in §5.4); since not all servers will be relocated in practice, the average circuit duty cycle for all transmissions can be as high as 99.83%. Fig. 11(b) shows the distribution of flow path lengths for RDC, C-Through, and NBLK. (Two C-Through settings have the same distribution; NBLK, 4:1-o.s., and RotorNet also have the same distribution). An intra-rack flow has path length 2 in all networks; and an inter-rack flow has path length 4 in RDC, NBLK, and 4:1-o.s.; the path length could vary in C-Through—3 for the circuit path and 4 for the normal packet-switched path. Overall, RDC localizes more than 70% of the inter-rack traffic and achieves an average path length of $0.75\times$ of C-Through and $0.65\times$ of NBLK.

5.3 Packaging, power, and capital cost

Packaging. Fig. 12 shows the packaging design of an RDC pod, which is somewhat different from that of a traditional pod. RDC has a central switch rack dedicated to hosting ToRs, *agg.* switches, and OCSes. Server racks are connected to OCSes via fiber bundles to reduce wiring complexity. On the central rack, ToRs are connected to OCSes and *agg.* switches using short fibers and cables, respectively. *Agg.* switches provide similar connectivity to core switches outside the pod, just like in traditional data centers. To ensure that centralized switch placement has similar reliability as traditional switch placement, backup power supplies are employed. Similar to the existing modular data centers, RDC supports incremental expansion by adding RDC pods.

Power and capital cost modeling. We show that RDC is more economical by comparing the power and capital cost between RDC and NBLK, at 400 Gbps data rate. They both
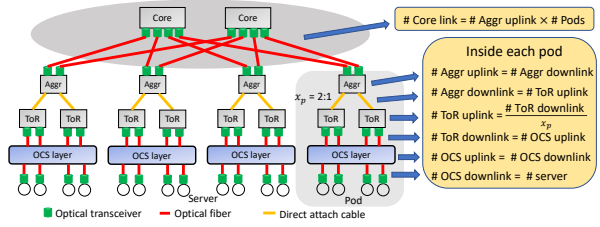
Figure 13: RDC example with detailed components, including the governing equations for power and capital cost model.

Components	Power (Watt)	Cost (USD)	Relative count RDC (x_p : 1)	NBLK
Ethernet port [16]	40.6	312.5	$1+4/x_p$	5
Optical transceiver [15]	10	799	$2+2/x_p$	4
Inter-rack fiber [20]	0	6.9	$1+1/x_p$	2
Intra-rack fiber [21]	0	4.9	1	0
DAC [17]	1.5	249	$1/x_p$	1
OCS port [8]	0.14	400 [52]	2	0

Table 1: Power/cost data and relative count of the components for RDC (x_p:1 o.s.) and NBLK at 400 Gbps.

consist of the following types of networking components: a) 400 Gbps Ethernet port, b) 400 Gbps Optical transceiver, c) inter-rack duplex single-mode fiber (average length 10m), d) intra-rack duplex single-mode fiber (average length 3m), e) 400 Gbps Direct Attach Cables (average length 3m) and f) OCS port. NBLK network can use DAC to directly connect the server-ToR downlinks, while RDC needs fiber-optic cables along with optical transceivers both at the server and ToR ends to connect the OCSes in between.

We assume that RDC has an x_p : 1 oversubscription above the ToR level. Fig. 13 demonstrates a 4-pod RDC network (total 16 servers) with component-level details (where $x_p = 2$) and shows the governing equations to find the component counts across the network. Based on our modeling, given the number of servers and pods are the same, the relative component count for RDC and NBLK network only depends on x_p. Table 1 shows the recent power and cost values of different components along with their relative count for RDC and NBLK network (400 Gbps). On one hand, the power consumption values of the network components are fundamental and well-documented in datasheets. On the other hand, the component cost can vary based on sales volume, and since we have no proprietary industry pricing figures, we do a "best-effort" calculation based on readily available retail pricing (in other words worst-case or no-discount pricing) for all components, so at least it is somewhat objective and unbiased. We consider $400 to be the OCS per-port cost, the worst-case price adopted from a recently reported article from Microsoft [52]. Readers should be aware of the limitation of this pricing assumption and take the capital cost results for general guidance only.

As shown in the governing equations in Fig. 13, an x_p : 1 RDC network with *s* servers have *s* ToR downlink ports, $\frac{s}{x_p}$ ToR uplink ports, $\frac{s}{x_p}$ *agg.* switch downlink ports, $\frac{s}{x_p}$ *agg.*
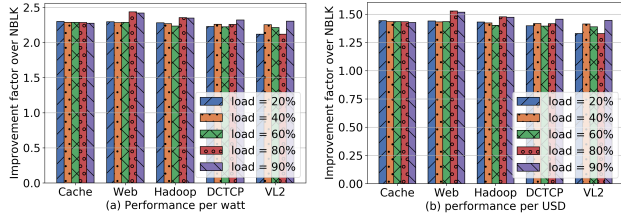
Figure 14: a) RDC (4:1-o.s.) has 2.1-2.4× improvements in performance per watt than NBLK at 400 Gbps data rate; b) RDC (4:1-o.s.) has 1.3-1.5× improvements in performance per dollar than NBLK at 400 Gbps data rate, assuming worst-case component pricing.

switch uplinks ports and $\frac{s}{x_p}$ core switch ports; leading to $(s + \frac{4s}{x_p})$ Ethernet ports in total. Consider a traditional NBLK (fat-tree) network with the same number of servers and pods, where the number of Ethernet switch ports at each layer is the same as the number of servers. This leads to a total of $5s$ (using $x_p = 1$ in RDC) Ethernet ports. Hence, the relative Ethernet port count is $(1 + \frac{4}{x_p})$ to 5 (see Table 1). Similar calculation can be applied to other components as well.

Power efficiency. A 4:1-o.s. RDC network consumes 2.29× less power than an NBLK network considering 400 Gbps data rate. RDC significantly improves the performance (median FCT) per watt compared to that of NBLK for diverse traffic patterns across different network loads, as shown in Fig. 14(a). We use five different production traces i.e., Cache [99], Web [99], Hadoop [99], DCTCP [29] and VL2 [62]. For median FCT, RDC has $2.1 \times -2.4 \times$ improvements in performance per watt compared to NBLK. RDC also significantly reduces the power consumption of the network because it requires fewer power-hungry packet switches in the core. The optical circuit switch at the RDC edge consumes very little power since it only directs the incoming photon beams using mirror rotation or diffraction.

Capital cost. We again emphasize that readers should take this "best-effort" cost analysis for general guidance only. A 4:1-o.s. RDC network costs 1.4× less than an NBLK network at 400 Gbps. Using the same five production traces, we observe that RDC has $1.3 \times -1.5 \times$ improvements in performance (median FCT) per dollar compared to NBLK, as shown in Fig. 14(b). We also estimate OCS per-port cost which would let 4:1-o.s. RDC has an equal performance per dollar as NBLK: it ranges from $1000-$1300.

5.4 RDC reconfigurations

To have a deeper understanding of RDC, we break down this analysis into the effectiveness study of the demand estimation algorithm, the non-disruptive control loop before the reconfiguration, and the hardware transient state during the reconfiguration.

Effectiveness of demand estimation algorithm To show the effectiveness of our demand estimation algorithm, we examine how our heuristic interacts with consecutive topology
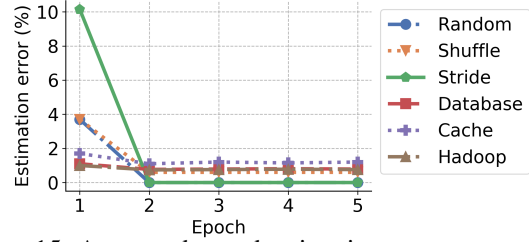
Figure 15: Average demand estimation error over multiple consecutive epochs (epoch duration: 10s).

reconfigurations, by an in-depth study of traffic localization.

We use the same packet-level simulation as §5.2 to illustrate this demand estimation technique. For each simulation, RDC performs traffic localization and reconfigures the topology once every 10s according to the algorithm detailed in §4.3. The senders and receivers of elastic flows are determined based on a chosen traffic pattern, while non-elastic flows are generated with randomly chosen senders and receivers with a data rate < 10Mbps. The ratio between the number of non-elastic and elastic flows is 10:1. Besides the three trace-derived traffic patterns – Cache, Web, and Hadoop, we also test three synthetic traces as follows. Each traffic pattern remains the same throughout the simulation.

Random: Each host i sends a flow to one of the other hosts with uniform random probability;

Shuffle: Each host i sends to a set of 31 other hosts with indexes $(i + j * 16)\% num\_hosts, j \in [1..31]$;

Stride: Each host i sends a flow to another host with index $(i + 32)\% num\_hosts$;

Fig. 15 shows the average demand estimation errors over five consecutive reconfiguration epochs for all server pairs. We observe that while the initial demand estimation errors can be moderately high (10%), the errors decrease as the network reconfigures to adapt to the traffic pattern in subsequent epochs. In the first epoch, many elastic flows congest the oversubscribed network core. As a result, their flow counters can be small and they could be misidentified as non-elastic flows. However, as RDC adapts the topology to localize the identified elastic flows, fewer elastic flows are transmitted across racks, congestion in the network core is reduced, and thus more elastic flows are correctly identified. For example, because the elastic flows in the stride pattern are eventually all localized within racks, the demand estimation errors for these elastic flows drop to nearly zero. Therefore, we can see that the Hedera technique is well-suited to RDC—reconfiguring the topology to suit the traffic patterns helps improve the accuracy of demand estimates for the next epoch.

Non-disruptive control loop. Next, we evaluate the latency of the RDC control loop, which includes four components: 1) collecting flow counters, 2) estimating traffic demands, 3) computing new topologies, and 4) modifying forwarding rules. This latency will affect how fast RDC can respond to changing traffic patterns. Note that the reactive RDC uses all four components; for proactive RDC using traffic demand matrix,

#Racks	4		8		16		32	
	TL	ULB	TL	ULB	TL	ULB	TL	ULB
Counter collection	10.6	2.3	21.3	2.6	42.6	3.4	85.1	4.5
Demand estimation	10.8	0.7	24.9	1.1	80.6	1.3	310.6	1.7
Topo. computation	7.8	0.1	28.2	0.1	40.3	0.3	69.3	0.6
Rule installation	32.5	30.6	45.6	30.8	75.6	41.4	147.6	70.6
Proactive - Command	**32.5**	**30.6**	**45.6**	**30.8**	**75.6**	**41.4**	**147.6**	**70.6**
Proactive - Demand	**40.3**	**30.7**	**73.8**	**30.9**	**115.9**	**41.7**	**216.9**	**71.2**
Reactive	**61.7**	**33.7**	**120**	**34.6**	**239.1**	**46.4**	**612.6**	**77.4**

Table 2: Control loop latency breakdown (ms) for traffic localization (TL) and uplink load-balancing (ULB).

only steps 3 & 4 will be executed; for proactive RDC with direct configuration command, only the last step is required.

To obtain these results, we ran a set of experiments using different numbers of racks, with 32 servers per rack, using the traffic patterns from the Facebook traces. The ToR switches are connected to a single *agg.* switch. Since our testbed only has four ToR switches, we emulated more ToR switches using servers and ensured that each server has the same latency for collecting counters and installing routing rules as a physical ToR switch. The number of forwarding rules to be installed is bounded by 32 for the ToR switches and $32 \times$ #racks for the *agg.* switch. And the number varies depending on the traffic patterns and may be different across switches. The overall rule installation delay is determined by the slowest switch, which has the most number of changes.

Table 2 breaks down the control loop latency for traffic localization (TL) and uplink load-balancing (ULB) use cases. Overall, reactive RDC's non-disruptive control loop latency before reconfiguration is 612.6ms for TL and 77.4ms for ULB, which are on similar timescales with state-of-the-art traffic engineering techniques [38]. Whereas, proactive RDC can reduce this control loop delay to 147.6 ms and 70.6 ms respectively. Since RDC aims to reconfigure the network at large timescales (e.g., seconds or longer), this control loop is efficient enough to be practical. Note that all the above numbers are obtained with our own testbed. With the cutting-edge high-performance switch hardware [9, 16, 18], the latency can be further reduced to support more frequent reconfiguration.

Reconfiguration transient state. It is important to observe that a circuit reconfiguration in RDC happens only when needed, and for the vast majority of the time, circuits are continuously active. When a reconfiguration happens to a circuit, a transient disruption to that circuit does occur. For example, AWGR and star-coupler-based OCSes are becoming popular as tunable lasers with sub-nanosecond wavelength switching are being fabricated [33, 35, 48, 49, 58, 77]. Considering 400 Gbps link speed and 1 ns of switching delay, only 50 bytes of traffic will be buffered or dropped during the transient phase. Also, 2D-MEMS based OCSes are available, having a reconfiguration delay of few microseconds [96]. Even with a relatively slow OCS in our testbed, our experiments show that RDC provides large performance benefits.

6 Related Work

Various DCN proposals recognize the need for serving dynamic workloads and provision bandwidth on demand with reconfigurable topologies. It can be achieved by adding extra bandwidth to the network by creating ad hoc links at runtime [53, 74, 80, 110, 118], but they mostly focus on providing reconfigurable topology at the rack level, assuming skewed inter-rack traffic. RDC, however, alleviates the reliance on such an assumption and achieves higher performance without adding extra bandwidth. Another line of work constructs an all-connected flexible network core with a high capacity [32, 44, 84–86, 96], but they mostly focus on rack-level rather than edge reconfigurability. Flat-tree [115] is an architecture proposal with partial edge-level reconfigurability, which enables DC-wide reconfigurability by dynamically changing the topology between Clos [26] and random graph [106]. However, the topology modes are limited and only suitable for generally expected workload patterns, e.g., rack-, pod-, or DC-local. Our workshop paper [111] does not contain a detailed design, implementation, or evaluation.

Besides architectural solutions, there are also numerous works that improve flow performance by optimizing task placements. For instance, Sinbad [45] selectively chooses data transfer destinations to avoid network congestion; Shuffle-Watcher [25] attempts to localize the shuffle phase of MapReduce jobs to one or a few racks; Corral [72] jointly places input data and compute to reduce inter-rack traffic for recurring jobs. However, these works all have important drawbacks as they only optimize data transfer for one or two stages of job executions. As we noted before, the traffic pattern may change in different stages of a job's lifetime. Also, there is a set of research projects that improve network performance at the upper layers in the stack. Optimized transport protocols (e.g., DCTCP [28], MPTCP [97]) and traffic engineering techniques (e.g., Hedera [27], MicroTE [38], Varys [47]) can improve flow performance for many applications.

7 Conclusion

The rackless data center (RDC) is a novel network architecture that logically removes the static rack boundaries, using circuit switching to achieve topological reconfigurability at the edge. In this architecture, servers in different physical racks can be grouped into the same locality group at runtime based on traffic patterns. By co-designing the network architecture and the control systems, RDC can benefit a wide range of realistic data center workloads. Our evaluations with testbed and simulation setups show that RDC leads to substantial performance benefits for real-world applications.

Acknowledgment

References

[1] Top of rack vs end of row data center designs. http://bradhedlund.com/2009/04/05/top-of-rack-vs-end-of-row-data-center-designs/, 2009.

[2] S320 photonic switch hardware user manual. http://www.calient.net/wp-content/uploads/downloads/2013/04/CALIENT-S-Series-Photonic-Switch-Hardware-User-Manual-Rev-A-460xxx-00-v10.pdf, 2012.

[3] Introducing data center fabric, the next-generation facebook data center network. https://code.fb.com/production-engineering/introducing-data-center-fabric-the-next-\generation-facebook-data-center-network, 2014.

[4] Facebook network analytics data sharing. https://www.facebook.com/groups/1144031739005495/, 2016.

[5] Apache hadoop: Fair scheduler. https://hadoop.apache.org/docs/r2.7.4/hadoop-yarn/hadoop-yarn-site/FairScheduler.html, 2017.

[6] Sailing through the data deluge. https://rockleyphotonics.com/wp-content/uploads/2019/02/Rockley-Photonics-Sailing-through-the-Data-Deluge.pdf., 2019.

[7] 25.6 tb/s strataxgs broadcom tomahawk 4 ethernet switch series. https://www.broadcom.com/products/ethernet-connectivity/switching/ strataxgs/bcm56990-series, 2020.

[8] 320x320 3D MEMS optical circuit switch. https://www.calient.net/products/edge640-optical-circuit-switch/, 2020.

[9] 32*100Gbps Ethernet Switch. https://www.fs.com/products/107081.html, 2020.

[10] Apache hadoop. http://hadoop.apache.org, 2020.

[11] Apache hadoop: Capacity scheduler. https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html, 2020.

[12] Apache hadoop yarn project. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html, 2020.

[13] Hdfs architecture. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html, 2020.

[14] Specifying data center it pod architectures. https://www.apc.com/salestools/WTOL-AHAPRN/WTOL-AHAPRN_R0_EN.pdf, 2020.

[15] 100G PAM4 850nm 100m optical transceiver module. https://www.fs.com/products/93264.html, 2021.

[16] 32*400Gbps Ethernet Switch. https://www.fs.com/products/96982.html, 2021.

[17] 400G QSFP-DD Passive Direct Attach Copper Twinax Cable (3m). https://www.fs.com/products/82454.html, 2021.

[18] Barefoot tofino. https://www.barefootnetworks.com/products/brief-tofino, 2021.

[19] Core and pod data center design. http://go.bigswitch.com/rs/974-WXR-561/images/Core-and-Pod%20Overview.pdf, 2021.

[20] Duplex single mode optical fiber cable (10m). https://www.fs.com/products/40203.html, 2021.

[21] Duplex single mode optical fiber cable (3m). https://www.fs.com/products/40193.html, 2021.

[22] Ibm prefabricated modular data center. https://www.ibm.com/us-en/marketplace/prefabricated-modular-data-center, 2021.

[23] Memcached. https://memcached.org, 2021.

[24] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

[25] Faraz Ahmad, Srimat T Chakradhar, Anand Raghunathan, and TN Vijaykumar. Shufflewatcher: Shuffle-aware scheduling in multi-tenant mapreduce clusters. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 1–13, 2014.

[26] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 63–74. ACM, 2008.

[27] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: dynamic flow scheduling for data center networks. In *NSDI*, volume 10, pages 89–92, 2010.

[28] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 63–74, 2010.

[29] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 conference*, pages 63–74, 2010.

[30] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is more: trading a little bandwidth for ultra-low latency in the data center. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation NSDI 12)*, pages 253–266, 2012.

[31] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.

[32] Paraskevas Bakopoulos, Konstantinos Christodoulopoulos, Giada Landi, Muzzamil Aziz, Eitan Zahavi, Domenico Gallico, Richard Pitwon, Konstantinos Tokas, Ioannis Patronas, Marco Capitani, et al. Nephele: An end-to-end scalable and dynamically reconfigurable optical architecture for application-aware sdn cloud data centers. *IEEE Communications Magazine*, 56(2):178–188, 2018.

[33] Hitesh Ballani, Paolo Costa, Raphael Behrendt, Daniel Cletheroe, Istvan Haller, Krzysztof Jozwik, Fotini Karinou, Sophie Lange, Kai Shi, Benn Thomsen, et al. Sirius: A flat datacenter network with nanosecond optical switching. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 782–797, 2020.

[34] Hitesh Ballani, Paolo Costa, Istvan Haller, Krzysztof Jozwik, Kai Shi, Benn Thomsen, and Hugh Williams. Bridging the last mile for optical switching in data centers. In *Optical Fiber Communication Conference*, pages W1C–3. Optical Society of America, 2018.

[35] Joshua L Benjamin, Thomas Gerard, Domaniç Lavery, Polina Bayvel, and Georgios Zervas. Pulse: optical circuit switched data center architecture operating at nanosecond timescales. *Journal of Lightwave Technology*, 38(18):4906–4921, 2020.

[36] Theophilus Benson, Aditya Akella, and David A Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280, 2010.

[37] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Understanding data center traffic characteristics. In *Proceedings of the 1st ACM workshop on Research on enterprise networking*, pages 65–72. ACM, 2009.

[38] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Microte: Fine grained traffic engineering for data centers. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*, page 8. ACM, 2011.

[39] Sergey Blagodurov, Alexandra Fedorova, Evgeny Vinnik, Tyler Dwyer, and Fabien Hermenier. Multiobjective job placement in clusters. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2015.

[40] Peter Bodík, Ishai Menache, Mosharaf Chowdhury, Pradeepkumar Mani, David A Maltz, and Ion Stoica. Surviving failures in bandwidth-constrained datacenters. *ACM SIGCOMM Computer Communication Review*, 42(4):431–442, 2012.

[41] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. {TAO}: Facebook's distributed data store for the social graph. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60, 2013.

[42] Marco Canini, Petr Kuznetsov, Dan Levin, and Stefan Schmid. A distributed and robust sdn control plane for transactional network updates. In *2015 IEEE conference on computer communications (INFOCOM)*, pages 190–198. IEEE, 2015.

[43] Andromachi Chatzieleftheriou, Sergey Legtchenko, Hugh Williams, and Antony Rowstron. Larry: Practical network reconfigurability in the data center. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 141–156, 2018.

[44] Kai Chen, Ankit Singla, Atul Singh, Kishore Ramachandran, Lei Xu, Yueping Zhang, Xitao Wen, and Yan Chen. Osa: An optical switching architecture for data center networks with unprecedented flexibility. *IEEE/ACM Transactions on Networking*, 22(2):498–511, 2014.

[45] Mosharaf Chowdhury, Srikanth Kandula, and Ion Stoica. Leveraging endpoint flexibility in data-intensive clusters. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 231–242. ACM, 2013.

[46] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I Jordan, and Ion Stoica. Managing data transfers in computer clusters with orchestra. *ACM SIGCOMM Computer Communication Review*, 41(4):98–109, 2011.

[47] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient coflow scheduling with varys. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 443–454, 2014.

[48] Kari Clark, Hitesh Ballani, Polina Bayvel, Daniel Cletheroe, Thomas Gerard, Istvan Haller, Krzysztof Jozwik, Kai Shi, Benn Thomsen, Philip Watts, et al. Sub-nanosecond clock and data recovery in an optically-switched data centre network. In *2018 European Conference on Optical Communication (ECOC)*, pages 1–3. IEEE, 2018.

[49] Kari A Clark, Daniel Cletheroe, Thomas Gerard, Istvan Haller, Krzysztof Jozwik, Kai Shi, Benn Thomsen, Hugh Williams, Georgios Zervas, Hitesh Ballani, et al. Synchronous subnanosecond clock and data recovery for optically switched data centres using clock phase caching. *Nature Electronics*, 3(7):426–433, 2020.

[50] Sushovan Das, Weitao Wang, and TS Ng. Towards all-optical circuit-switched datacenter network cores: The case for mitigating traffic skewness at the edge. In *ACM SIGCOMM 2021 Workshop on Optical Systems (OptSys' 21)*, 2021.

[51] Mauro Dell'Amico and Silvano Martello. Bounds for the cardinality constrained p cmax problem. *Journal of Scheduling*, 4(3):123–138, 2001.

[52] Vojislav Dukic, Ginni Khanna, Christos Gkantsidis, Thomas Karagiannis, Francesca Parmigiani, Ankit Singla, Mark Filer, Jeffrey L Cox, Anna Ptasznik, Nick Harland, et al. Beyond the mega-data center: networking multi-data center regions. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 765–781, 2020.

[53] Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Hajabdolali Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Helios: a hybrid electrical/optical switch architecture for modular data centers. *ACM SIGCOMM Computer Communication Review*, 40(4):339–350, 2010.

[54] G.C Fox, S.W Otto, and A.J.G Hey. Matrix algorithms on a hypercube i: Matrix multiplication. *Parallel Computing*, 4(1):17 – 31, 1987.

[55] Rohan Gandhi, Hongqiang Harry Liu, Y Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. Duet: Cloud scale load balancing with hardware and software. *ACM SIGCOMM Computer Communication Review*, 44(4):27–38, 2014.

[56] Peter X Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 249–264, 2016.

[57] Thomas Gerard, Kari Clark, Adam Funnell, Kai Shi, Benn Thomsen, Philip Watts, Krzysztof Jozwik, Istvan Haller, Hugh Williams, Paolo Costa, et al. Fast and uniform optically-switched data centre networks enabled by amplitude caching. In *2021 Optical Fiber Communications Conference and Exhibition (OFC)*, pages 1–3. IEEE, 2021.

[58] Thomas Gerard, Christopher Parsonson, Zacharaya Shabka, Polina Bayvel, Domaniç Lavery, and Georgios Zervas. Swift: Scalable ultra-wideband subnanosecond wavelength switching for data centre networks. *arXiv preprint arXiv:2003.05489*, 2020.

[59] Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Nikhil Devanur, Janardhan Kulkarni, Gireeja Ranade, Pierre-Alexandre Blanche, Houman Rastegarfar, Madeleine Glick, and Daniel Kilper. Projector: Agile reconfigurable data center interconnect. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 216–229. ACM, 2016.

[60] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Nsdi*, volume 11, pages 24–24, 2011.

[61] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. *ACM SIGCOMM Computer Communication Review*, 44(4):455–466, 2014.

[62] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, pages 51–62, 2009.

[63] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 649–667, 2017.

[64] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. Bcube: a high performance, server-centric network architecture for modular data centers. *ACM SIGCOMM Computer Communication Review*, 39(4):63–74, 2009.

[65] Chuanxiong Guo, Haitao Wu, Kun Tan, Lei Shi, Yongguang Zhang, and Songwu Lu. Dcell: a scalable and fault-tolerant network structure for data centers. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 75–86. ACM, 2008.

[66] Navid Hamedazimi, Zafar Qazi, Himanshu Gupta, Vyas Sekar, Samir R Das, Jon P Longtin, Himanshu Shah, and Ashish Tanwer. Firefly: A reconfigurable wireless data center fabric using free-space optics. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 319–330. ACM, 2014.

[67] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 29–42, New York, NY, USA, 2017. ACM.

[68] Keqiang He, Junaid Khalid, Aaron Gember-Jacobson, Sourav Das, Chaithan Prakash, Aditya Akella, Li Erran Li, and Marina Thottan. Measuring control plane latency in sdn-enabled switches. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, page 25. ACM, 2015.

[69] Christian E Hopps. Analysis of an equal-cost multi-path algorithm. 2000.

[70] Chang-Hong Hsu, Qingyuan Deng, Jason Mars, and Lingjia Tang. Smoothoperator: Reducing power fragmentation and improving power utilization in large-scale datacenters. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 535–548, New York, NY, USA, 2018. ACM.

[71] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 261–276, 2009.

[72] Virajith Jalaparti, Peter Bodik, Ishai Menache, Sriram Rao, Konstantin Makarychev, and Matthew Caesar. Network-aware scheduling for data-parallel jobs: Plan when you can. *ACM SIGCOMM Computer Communication Review*, 45(4):407–420, 2015.

[73] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. Dynamic scheduling of network updates. *ACM SIGCOMM Computer Communication Review*, 44(4):539–550, 2014.

[74] Srikanth Kandula, Jitendra Padhye, and Paramvir Bahl. Flyways to de-congest data center networks. 2009.

[75] Robert Krauthgamer, Joseph Naor, and Roy Schwartz. Partitioning graphs into balanced components. In *Proceedings of the twentieth annual ACM-SIAM symposium on Discrete algorithms*, pages 942–949. SIAM, 2009.

[76] Maciej Kuźniar, Peter Perešíni, and Dejan Kostić. What you need to know about sdn flow tables. In *International Conference on Passive and Active Network Measurement*, pages 347–359. Springer, 2015.

[77] Sophie Lange, Arslan S Raja, Kai Shi, Maxim Karpov, Raphael Behrendt, Daniel Cletheroe, Istvan Haller, Fotini Karinou, Xin Fu, Junqiu Liu, et al. Sub-nanosecond optical switching using chip-based soliton microcombs. In *Optical Fiber Communication Conference*, pages W2A–4. Optical Society of America, 2020.

[78] Dominique LaSalle and George Karypis. Multithreaded graph partitioning. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 225–236. IEEE, 2013.

[79] T Li, B Cole, P Morton, and D Li. Rfc2281: Cisco hot standby router protocol (hsrp), 1998.

[80] He Liu, Feng Lu, Alex Forencich, Rishi Kapoor, Malveeka Tewari, Geoffrey M. Voelker, George Papen, Alex C. Snoeren, and George Porter. Circuit switching under the radar with reactor. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 1–15, Seattle, WA, 2014. USENIX Association.

[81] Hongqiang Harry Liu, Xin Wu, Ming Zhang, Lihua Yuan, Roger Wattenhofer, and David Maltz. zupdate: Updating data center networks with zero loss. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 411–422. ACM, 2013.

[82] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. F10: A fault-tolerant engineered network. In *Presented as part of the 10th*

USENIX Symposium on Networked Systems Design and Implementation (NSDI 13), pages 399–412, 2013.

[83] Vincent Liu, Danyang Zhuo, Simon Peter, Arvind Krishnamurthy, and Thomas Anderson. Subways: A case for redundant, inexpensive data center edge links. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, page 27. ACM, 2015.

[84] Yunpeng James Liu, Peter Xiang Gao, Bernard Wong, and Srinivasan Keshav. Quartz: a new design element for low-latency dcns. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 283–294. ACM, 2014.

[85] William M. Mellette, Rajdeep Das, Yibo Guo, Rob McGuinness, Alex C. Snoeren, and George Porter. Expanding across time to deliver bandwidth efficiency and low latency. *arXiv e-prints*, page arXiv:1903.12307, Mar 2019.

[86] William M Mellette, Rob McGuinness, Arjun Roy, Alex Forencich, George Papen, Alex C Snoeren, and George Porter. Rotornet: A scalable, low-complexity, optical datacenter network. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 267–280. ACM, 2017.

[87] Xiaoqiao Meng, Vasileios Pappas, and Li Zhang. Improving the scalability of data center networks with traffic-aware virtual machine placement. In *2010 Proceedings IEEE INFOCOM*, pages 1–9. IEEE, 2010.

[88] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 15–28. ACM, 2017.

[89] Wil Michiels, Jan Korst, Emile Aarts, and Jan Van Leeuwen. Performance ratios for the differencing method applied to the balanced number partitioning problem. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 583–595. Springer, 2003.

[90] Tal Mizrahi and Yoram Moses. Time4: Time for sdn. *IEEE Transactions on Network and Service Management*, 13(3):433–446, 2016.

[91] Samuel K Moore. Another step toward the end of moore's law: Samsung and tsmc move to 5-nanometer manufacturing-[news]. *IEEE Spectrum*, 56(6):9–10, 2019.

[92] Mihir Nanavati, Jake Wires, and Andrew Warfield. Decibel: Isolation and sharing in disaggregated {Rack-Scale} storage. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 17–33, 2017.

[93] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, 2013.

[94] Vlad Nitu, Boris Teabe, Alain Tchana, Canturk Isci, and Daniel Hagimont. Welcome to zombieland: practical and energy-efficient memory disaggregation in a datacenter. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–12, 2018.

[95] Rina Panigrahy, Kunal Talwar, Lincoln Uyeda, and Udi Wieder. Heuristics for vector bin packing. *research. microsoft. com*, 2011.

[96] George Porter, Richard Strong, Nathan Farrington, Alex Forencich, Pang Chen-Sun, Tajana Rosing, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Integrating microsecond circuit switching into the data center. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 447–458, New York, NY, USA, 2013. ACM.

[97] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving datacenter performance and robustness with multipath tcp. *ACM SIGCOMM Computer Communication Review*, 41(4):266–277, 2011.

[98] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. *ACM SIGCOMM Computer Communication Review*, 42(4):323–334, 2012.

[99] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the social network's (datacenter) network. *SIGCOMM Comput. Commun. Rev.*, 45(4):123–137, August 2015.

[100] Liron Schiff, Stefan Schmid, and Petr Kuznetsov. In-band synchronization for distributed sdn control planes. *ACM SIGCOMM Computer Communication Review*, 46(1):37–43, 2016.

[101] Tae Joon Seok, Niels Quack, Sangyoon Han, Wencong Zhang, Richard S Muller, and Ming C Wu. Reliability study of digital silicon photonic mems switches. In *2015 IEEE 12th International Conference on Group IV Photonics (GFP)*, pages 205–206. IEEE, 2015.

[102] Tae Joon Seok, Niels Quack, Sangyoon Han, Wencong Zhang, Richard S Muller, and Ming C Wu. Reliability study of digital silicon photonic mems switches. In *Group IV Photonics (GFP), 2015 IEEE 12th International Conference on*, pages 205–206. IEEE, 2015.

[103] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. {LegoOS}: A disseminated, distributed {OS} for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 69–87, 2018.

[104] Vishal Shrivastav, Asaf Valadarsky, Hitesh Ballani, Paolo Costa, Ki Suh Lee, Han Wang, Rachit Agarwal, and Hakim Weatherspoon. Shoal: A network architecture for disaggregated racks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 255–270, Boston, MA, 2019. USENIX Association.

[105] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. *ACM SIGCOMM computer communication review*, 45(4):183–197, 2015.

[106] Ankit Singla, Chi-Yao Hong, Lucian Popa, and Philip Brighten Godfrey. Jellyfish: Networking data centers, randomly. In *NSDI*, volume 12, pages 1–6, 2012.

[107] Rob Stone, Ruby Chen, Jeff Rahn, Srinivas Venkataraman, Xu Wang, Katharine Schmidtke, and James Stewart. Co-packaged optics for data center switching. In *2020 European Conference on Optical Communications (ECOC)*, pages 1–3. IEEE, 2020.

[108] Xiongchao Tang, Haojie Wang, Xiaosong Ma, Nosayba El-Sayed, Jidong Zhai, Wenguang Chen, and Ashraf Aboulnaga. Spread-n-share: improving application performance and cluster throughput with resource-aware job placement. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2019.

[109] Meg Walraed-Sullivan, Amin Vahdat, and Keith Marzullo. Aspen trees: balancing data center fault tolerance, scalability and cost. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pages 85–96, 2013.

[110] Guohui Wang, David G Andersen, Michael Kaminsky, Konstantina Papagiannaki, TS Ng, Michael Kozuch, and Michael Ryan. c-through: Part-time optics in data centers. In *ACM SIGCOMM Computer Communication Review*, volume 40, pages 327–338. ACM, 2010.

[111] Dingming Wu, Weitao Wang, Ang Chen, and TS Ng. Say no to rack boundaries: Towards a reconfigurable pod-centric dcn architecture. In *Proceedings of the 2019 ACM Symposium on SDN Research*, pages 112–118. ACM, 2019.

[112] Dingming Wu, Yiting Xia, Xiaoye Steven Sun, Xin Sunny Huang, Simbarashe Dzinamarira, and TS Eugene Ng. Masking failures from application performance in data center networks with shareable backup. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 176–190, 2018.

[113] Xin Wu, Daniel Turner, Chao-Chih Chen, David A Maltz, Xiaowei Yang, Lihua Yuan, and Ming Zhang. Netpilot: automating datacenter network failure mitigation. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 419–430. ACM, 2012.

[114] Yiting Xia, Xin Sunny Huang, and T. S. Eugene Ng. Stop rerouting!: Enabling sharebackup for failure recovery in data center networks. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets-XVI, pages 171–177, New York, NY, USA, 2017. ACM.

[115] Yiting Xia, Xiaoye Steven Sun, Simbarashe Dzinamarira, Dingming Wu, Xin Sunny Huang, and TS Eugene Ng. A tale of two topologies: Exploring convertible data center network architectures with flat-tree. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 295–308, 2017.

[116] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, pages 265–278, 2010.

[117] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy Katz. Detail: reducing the flow completion time tail in datacenter networks. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 139–150. ACM, 2012.

[118] Xia Zhou, Zengbin Zhang, Yibo Zhu, Yubo Li, Saipriya Kumar, Amin Vahdat, Ben Y Zhao, and Haitao Zheng. Mirror mirror on the ceiling: Flexible wireless links for data centers. *ACM SIGCOMM CCR*, 42(4):443–454, 2012.

[119] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Klaus-Tycho Förster, Arvind Krishnamurthy, and Thomas Anderson. Understanding and mitigating packet corruption in data center networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 362–375, 2017.

[120] Christopher Zimmer, Saurabh Gupta, Scott Atchley, Sudharshan S Vazhkudai, and Carl Albing. A multifaceted approach to job placement for improved performance on extreme-scale systems. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1015–1025. IEEE, 2016.

A Appendix

This appendix includes more discussions and results.

A.1 The RDC 0/1 updates

Instead of changing packet version, in RDC a switch performs binary changes from VLAN tagging packets to not, and from not VLAN tagging packets to tagging. Assume packets are in VLAN tagging mode before the change and there is a single VLAN tagging rule at the ingress switch for all packets. We first install the new set of rules with lower priority that matches only on destination IPs, note that the more general matching rules always have lower priority. Then, we remove the VLAN tagging rule. The untagged packets in the transient state can immediately match against the new set of rules. Similarly, if packets are not in VLAN tagging mode before the update, we first install the new set of rules matches on both VLAN tag and destination IPs and then install a single VLAN tagging rule for all packets.

Fig. 16 illustrates the update mechanism in RDC, which we call 0/1 update. It uses an example of forwarding state updates on an OpenFlow ToR switch, which has 4 ports. Ports 1 and 2 are connected to servers, ports 3 and 4 are connected to the *agg.* switches. Packet versions are encoded in the VLAN tag. Before the update, packets are first matched against a VLAN table that tags packets with a VLAN ID. Those tagged packets are then matched against the old rules in the forwarding table. During the transient state of rule updating, packets become untagged and can thus immediately match against the new rules without being dropped. The instructions of the forwarding table direct packets to the group table where packets are either directly sent out via an output port or get load-balanced over multiple output ports using the *select* group type. Similarly, an update from the not-tagging mode to the tagging mode also causes no packet loss.

MatchFields	Instruction
ingressPort	set-field
*	set vlanID 1/None

vlan table

GroupID	GroupType	ActionBuckets
1	indirect	out_port 1
2	indirect	out_port 2
3	select	out_port 3: 0.5 out_port 4: 0.5

group table

MatchFields		Priority	Instruction	
vlanID	dstIp		GroupID	
1	d1	2	1	old rules
1	d2	2	2	
*	d1	1	2	new rules
*	d2	1	1	
*	*		3	

forwarding table
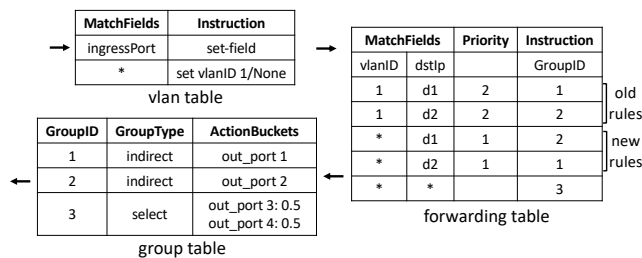
Figure 16: An example of RDC's 0/1 rule update on an OpenFlow-enabled ToR switch.

A.2 Use case: Uplink load-balancing

In §4, we have discussed four use cases for RDC. Among these, uplink load balancing is another reactive RDC algorithm. We discuss this use case in more detail, including the data collection, bias mitigation, and control algorithm. The proactive mode (Use case 3) is simply driven by applications,

and the mixed optimization (Use case 4) is also case-specific, so we focus on the reactive algorithm for Use case 2.

Traffic data collection. RDC maintains flow counters on ToRs to monitor the amount of traffic that each server has sent outside the pod. We assume each RDC pod has a unique ID, e.g., an IP address prefix shared by all servers in the pod. Counters are only installed and updated for inter-pod traffic. This can be implemented in the switch using two separate flow tables. The first flow table matches on the destination IP prefix and has only one rule matching the switch's own pod ID. If the first table misses, the second table then matches the 5-tuple and updates the associated counters. Otherwise, the packet skips the second table and goes to the forwarding table. By default, a miss on the second table will not result in packet loss, but a *go-to* action to the rest of the switch pipeline, which avoids traffic disruption when the counter rules change.

Demand estimation. We use a similar technique to estimate the true demand of servers in bottlenecked racks assuming they fair-share the uplink bandwidth. The estimates are obtained by first aggregating the flow counters for each server and then scaling up the per-server demand to reach an aggregate uplink throughput as if the rack is not oversubscribed. We only apply this technique to racks that have been bottlenecked in the collection period to prevent idle racks from being mistakenly treated as hot. This technique keeps the relative order of server traffic load but brings larger quantitative differences among servers, guiding our algorithm to compute better topologies.

Algorithm. For 1-CS RDC, we view the uplink load-balancing problem as a balanced graph partition problem; for multi-CS RDC, we use a heuristic algorithm to obtain the reconfiguration plan. The details of the above two algorithms are included in §A.4.

A.3 Hedera demand estimation algorithm

The pseudocode is shown in Algorithm 1. M is the demand matrix, H is the set of hosts in the network. e_S is the equal share rate of the flows, d_T is the total demand for the destination, and d_S is the demand limited by the sender, $f.rl$ is a flag for a receiver-limited flow, and $< src \rightarrow anydst >$ represents all the flows from the specific source host *src* to any destination host.

A general explanation for this algorithm is expanding the flow demand at the source host with the fair share, and then reducing the demand of some flows according to the capacity of the destination hosts. In each iteration, one or more flows will converge. Eventually, all the flows will converge after multiple iterations [27].

A.4 Topology optimization algorithm details

1. Problem formulation. Assume that the number of racks in a pod is m, each rack has n servers, and each pod has k CS switches to reallocate the server. To keep a record of which server is connected to which ToR switch, we use another

Algorithm 1: Hedera demand estimation [27]

Input: M: traffic matrix, H: the set of all hosts
Output: M: estimated demand matrix

1 **while** *some $M_{i,j}$ demand changed* **do**
2   **for** *host src $\in H$* **do**
3       $es \leftarrow \frac{1-\sum \text{converged flow demand}}{\text{unconverged flow number}}$,
      $flow \in< src \rightarrow anydst >$
4   **for** *flow $f \in< src \rightarrow anydst >$* **do**
5   **if** *f not converged* **then**
6             $M_{f.src,f.dst}.demand \leftarrow e_S$

7   **for** *host dst $\in H$* **do**
8   **for** *$f \in< anysrc \rightarrow dst >$* **do**
9          $f.rl \leftarrow true$
10          $d_T \leftarrow d_T + f.demand$
11          $n_R \leftarrow n_R + 1$
12   **if** *$d_T > 1$* **then**
13          $e_S \leftarrow \frac{1}{n_R}$
14   **while** *some f.rl was set to false* **do**
15             $n_R \leftarrow 0$
16   **for** *$f \in< anysrc \rightarrow dst > \& f.rl$* **do**
17   **if** *f.demand $< e_S$* **then**
18                   $d_S \leftarrow d_S + f.demand$
19                   $f.rl \leftarrow false$
20   **else**
21                   $n_R \leftarrow n_R + 1$
22             $es \leftarrow \frac{1-d_S}{n_R}$
23   **for** *$f \in< src \rightarrow dst > \& f.rl$* **do**
24             $M_{f.src,f.dst}.demand \leftarrow es$
25             $M_{f.src,f.dst}.converged \leftarrow true$

---

matrix $C[mn][m]$, if $C[i][j]$ is 1, then server i is connected to ToR j; the server and TOR are not connected if the value is 0. For a valid allocation of the servers, the first constraint is that one server should only be connected to one ToR:

$$\sum_{j=0}^{m-1} C[i][j] = 1, \forall i \in [0, mn) \tag{1}$$

The second constraint is because only a limited number of ports from each ToR are connected to every CS, which is $\frac{n}{k}$. Thus, among all the $\frac{mn}{k}$ servers connected to one CS, only $\frac{n}{k}$ of them can be connected to the same ToR switch:

$$\sum_{x=0}^{m-1} \sum_{y=0}^{\frac{n}{k}-1} C[x \cdot n + i * \frac{n}{k} + y][j] = \frac{n}{k}, \forall i \in [0, k), \forall j \in [0, m) \tag{2}$$

The goal for traffic localization is to localize the inter-rack

traffic within a pod as much as possible. Hence, the objective function is to maximize the total amount of localized traffic. Assume that the traffic demand matrix is $D[mn][mn]$, which covers all the server pairs in a pod. Only when two servers are connected to the same ToR, $C[x][j] \cdot C[y][j] = 1$, so that the following equation shows the amount of localized traffic demand:

$$\text{Maximize: } \sum_{x=0}^{mn-1} \sum_{y=0}^{mn-1} \sum_{j=0}^{m-1} C[x][j] \cdot C[y][j] \cdot D[x][y] \tag{3}$$

The goal for uplink load-balancing is to balance the load across all uplinks. Hence, we choose to minimize the maximum load of any uplink for the out-of-pod traffic. Assume that the out-of-pod traffic demand matrix is $U[mn]$. The objective function is:

$$\text{Minimize: } MAX\left\{ \sum_{i=0}^{mn-1} C[i][j] \cdot U[i] \right\}, j \in [0, m) \tag{4}$$

2. Heuristic traffic localization algorithm for 1-CS RDC. For RDC with only 1 circuit switch, the topology optimization problem will just become a graph partition problem. And the new objective function is that we want to partition the vertices (servers) in the graph into groups equally and let the edges (traffic demand) within the groups to be maximum. Assume the traffic demand is a graph $G = (E, V)$, where V is the vertex set (i.e., servers) and E is the edge set. The weight of an edge $e, w(e)$, is the traffic demand between the vertices. To simplify the computation, we do not distinguish the directions of traffic between a server pair, i.e., graph G is non-directional. Our goal is to partition the graph into subgraphs of equal numbers of vertices such that the weighted sum of cross-subgraph edges is minimized. We require partitions of the same size because each ToR must connect to the same fixed number of servers. The balanced graph partitioning problem is NP-hard, but high-quality, efficient heuristics have been proposed in a library *parmetis* [78]. Thus, for the traffic localization problem, we can set the objective to maximize the edge weights insides each group and use the BGP method to solve it.

3. Heuristic uplink load-balancing algorithm for 1-CS RDC. For RDC with only 1 circuit switch, the uplink load-balancing problem will also become a graph partition problem. Our formulation partitions mn number of servers $1, 2, ..., mn$ into m subsets $S_1, S_2, ...S_m$ such that each subset S_j has exactly n servers and the maximum cost of a subset, defined as $\max(\{c(S_j)\})$ is minimized, where $c(S_j) = \sum U[i](i \in S_j)$. Again, we require a balanced partition of the servers because each ToR must host the same number of servers. The problem is also NP-hard when $k > 2$ [51, 89]. We use the same high-quality and efficient heuristics, *parmetis*, to solve this problem by simply changing the objective function to balance the out-of-pod throughput for each group.

4. Heuristic traffic localization algorithm for multi-CS RDC. For RDC with multiple circuit switches, our heuristic firstly groups the servers under the same CS into m bundles equally and maximizes the traffic within each bundle, since all the servers within a bundle should be connected to the same ToR. After obtaining the bundles, for one CS, we only need to assign each of them to a different ToR switch, and the goal is to maximize the traffic demand among bundles under the same ToR switch. In total we have mk bundles, each bundle will be connected to one ToR switch, recorded as $BC[mk][m]$. Moreover, the bundles can be used to calculate an aggregated traffic demand matrix $BD[mk][mk]$. Thus, the simplified traffic localization algorithm can be presented as:

$$\sum_{j=0}^{m-1} BC[i][j] = 1, \forall i \in [0, mk) \tag{5}$$

$$\sum_{x=0}^{m-1} BC[x \cdot m + i][j] = 1, \forall i \in [0, m), \forall j \in [0, m) \tag{6}$$

$$\text{Maximize: } \sum_{x=0}^{mk-1} \sum_{y=0}^{mk-1} \sum_{j=0}^{m-1} BC[x][j] \cdot BC[y][j] \cdot BD[x][y] \tag{7}$$

5. Heuristic uplink load-balancing algorithm for multi-CS RDC. For the heuristic ULB algorithm, again the servers are grouped under the same CS into m bundles equally. And the objective function is to minimize the maximum out-of-pod traffic of each bundle. The idea behind this heuristic is that if each OCS gives balanced out-of-pod traffic to each ToR, then the total out-of-pod traffic from each ToR should also be balanced. The constraints remain the same. Thus, we divide the problem into many sub-problems, and each sub-problem focuses on the servers connected to the same OCS:

$$\sum_{j=0}^{m-1} C[i][j] = 1, \forall i \in [0, mn) \tag{8}$$

$$\sum_{x=0}^{m-1} \sum_{y=0}^{\frac{n}{k}-1} C[x \cdot n + i * \frac{n}{k} + y][j] = \frac{n}{k}, \forall i \in [0, k), \forall j \in [0, m) \tag{9}$$

$$\text{Minimize: } MAX \left\{ \sum_{r=0}^{m-1} \sum_{i=0}^{\frac{n}{k}-1} C[r * n + \frac{n}{k} * s + i][j] \cdot U[i] \right\}, \forall j \in [0, m) \tag{10}$$

Isolation Mechanisms for High-Speed Packet-Processing Pipelines

Tao Wang[†]   Xiangrui Yang[‡][*]   Gianni Antichi[**]   Anirudh Sivaraman[†]   Aurojit Panda[†]

[†]*New York University*   [‡]*National University of Defense Technology*
[**]*Queen Mary University of London*

Abstract

Data-plane programmability is now mainstream. As we find more use cases, deployments need to be able to run multiple packet-processing modules in a single device. These are likely to be developed by independent teams, either within the same organization or from multiple organizations. Therefore, we need isolation mechanisms to ensure that modules on the same device do not interfere with each other.

This paper presents Menshen, an extension of the Reconfigurable Match Tables (RMT) pipeline that enforces isolation between different packet-processing modules. Menshen is comprised of a set of lightweight hardware primitives and an extension to the open source P4-16 reference compiler that act in conjunction to meet this goal. We have prototyped Menshen on two FPGA platforms (NetFPGA and Corundum). We show that our design provides isolation, and allows new modules to be loaded without impacting the ones already running. Finally, we demonstrate the feasibility of implementing Menshen on ASICs by using the FreePDK45nm technology library and the Synopsys DC synthesis software, showing that our design meets timing at a 1 GHz clock frequency and needs approximately 6% additional chip area. We have open sourced the code for Menshen's hardware and software at https://isolation.quest/.

1 Introduction

Programmable network devices in the form of programmable switches [6, 15, 26] and smart network interface cards (SmartNICs) [10, 11, 44] are becoming commodity. Such devices allow the network infrastructure to provide its users additional services beyond packet forwarding, e.g., congestion control [41, 66], measurement [52], load balancing [62], in-network caches [60], and machine learning [72].

As network programmability matures, a single device will have to concurrently support *multiple* independently developed modules. This is the case for *networks in the public cloud* where tenants can provide packet-processing

---

[*]Work done at Queen Mary University of London
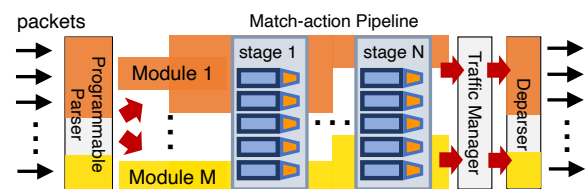
Figure 1: *The RMT architecture [36] typically consists of a programmable parser/deparser, match-action pipeline and traffic manager. Menshen provides isolation between RMT modules. In the figure we show resources allocated to module 1 and module m by shading them in the appropriate color.*

modules that are installed and run on the cloud provider's devices. Another example is when different teams in an organization write different modules, e.g., an in-networking caching module and a telemetry module.

Isolation is required to safely run multiple modules on a single device. Several prior projects have observed this need and proposed solutions targeting multicore network processors [50, 68], FPGA-based packet processors [63, 73, 77, 82], and software switches [53, 81]. However, thus far, high-speed pipelines such as RMT that are used in switch and NIC ASICs provide only limited support for isolation. For instance, the Tofino programmable switch ASIC [26] provides mechanisms to share stateful memory across modules but cannot share other resources, e.g., match-action tables [79].

Our goal with this paper is to lay out the requirements for isolation mechanisms on the RMT architecture that are applicable to all resources and then to design lightweight mechanisms that meet these requirements. As presented in Figure 1, the desired isolation mechanisms should guarantee that multiple modules can be allocated to different resources, and process packets in parallel without impacting each other. In brief (§2.1 elaborates), we seek isolation mechanisms that ensure that (1) one module's behavior (input, output, and internal state) is unaffected by another module; (2) one module can not affect another's throughput and/or latency; and (3) one module can not access RMT pipeline resources belonging to another. Given the high performance

requirements of RMT, we also seek mechanisms that are lightweight. Finally, the isolation mechanism should ensure that one module can be updated without disturbing any other modules and that the update process itself is quick.

The RMT architecture poses unique challenges for isolation because its pipeline design means that neither an OS nor a hypervisor can be used to enforce isolation.[1] This is because RMT is a *dataflow* or spatial hardware architecture [34, 39] with a set of instructions units continuously processing data (packets). This is in contrast to the Von Neumann architecture found on processors [27], where a program counter decides what instruction to execute next. As such, an RMT pipeline is closer in its hardware architecture to an FPGA or a CGRA [70] than a processor. This difference in architecture has important implications for isolation. The Von Neumann architecture supports a *time-sharing* approach to isolation (in the form of an OS/hypervisor) that runs different modules on the CPU successively by changing the program counter to point to the next instruction of the next module. We instead use *space-partitioning* to divide up the RMT pipeline's resources (e.g., match-action tables) across different modules.

Unfortunately, space partitioning is not a viable option for certain RMT resources because there are very few of them to be effectively partitioned across modules (e.g., match key extraction units (§3.1)). For such resources, we add additional hardware primitives in the form of small tables that store module-specific configurations for these resources. As a packet progresses through the pipeline, the packet's module identifier is used as an index into these tables to extract module-specific configurations before processing the packet according to the just extracted configuration. These primitives are similar to the use of *overlays* [3, 16] in embedded systems [1, 25] and earlier PCs [17]. They effectively allow us to bring in different configurations for the same RMT resource, in response to different packets from different modules.

Based on the ideas of space partitioning and overlays, we build a system, Menshen, for isolation on RMT pipelines. Specifically, Menshen makes the following contributions:

1. The use of space partitioning and overlays as techniques to achieve isolation when sharing an RMT pipeline across multiple modules.

2. A hardware design for an RMT pipeline that employs these techniques.

3. An implementation on 2 open-source FPGA platforms: the NetFPGA switch [84] and Corundum NIC [45].

4. A compiler based on the open-source P4-16 compiler [18] that supports multiple modules running on RMT, along with a system-level module to provide basic services (e.g., routing, multicast) to other modules.

5. An evaluation of Menshen using 8 modules—based on tutorial P4 programs, and the NetCache [60] and NetChain [59] research projects—showing that

Menshen meets our isolation requirements.

6. An ASIC analysis of the Menshen, which shows that our design can meet timing at 1 GHz (comparable to current programmable ASICs) with modest additional area relative to a baseline RMT design.

Overall, we find that Menshen adds modest overhead to an existing RMT pipeline in both FPGA and ASIC implementations (§5). Our main takeaway is that a small number of simple additions to RMT along with changes to the RMT compiler can provide inter-module isolation for a high-speed packet-processing pipeline. We have made Menshen's hardware design and software available under an open-source license at https://isolation.quest/ to enable further research into isolation mechanisms for high-speed pipelines.

2 The case for isolation

A single network device might host a measurement module [52], a forwarding module [74], an in-network caching [60] module, and an in-network machine-learning module [72]—each written by a different team in the same organization. It is important to isolate these modules from each other. This would prevent bugs in measurement, in-network caching, and in-network ML from causing network downtime. It would also ensure that memory for measuring per-flow stats [65] is separated from memory for routing tables, e.g., a sudden arrival of many new flows does not cause cached routes to be evicted from the data plane.

The packet-processing modules in question do not even have to be developed by teams in the same organization [79]. They could belong to different tenants sharing the same public cloud network. This would allow cloud providers to offer network data-plane programmability as a service to their tenants, similar to cloud CPU, GPU, and storage offerings today. Such a capability would allow tenants to customize network devices in the cloud to suit their needs.

2.1 Requirements for isolation mechanisms

For the rest of this paper, we will use the term module to refer to a packet-processing program that must be isolated from other such programs, regardless of whether the modules belong to different mutually distrustful tenants or to a single network operator. Importantly, modules can not call each other like functions, but are intended to isolate different pieces of functionality from each other—similar to processes. Based on our use cases above (§2), we want an inter-module isolation mechanisms that meet the requirements below:

1. **Behavior isolation.** The behavior of one module must not affect the behavior (i.e., input, output, computation and internal state) of another. This would prevent a faulty or malicious module from adversely affecting other modules. Further, one module should not be able to inspect the behavior of another module.

2. **Resource isolation.** A switch/NIC pipeline has multiple resources, e.g., static random-access memory (SRAM)

---

[1]An OS does run on the network device's control CPU, allowing isolation in the control plane. Our focus, instead, is on isolation in the data plane.

for exact matching and ternary content-addressable memory (TCAM) for ternary matching. Each module should be able to access only its assigned subset of the pipeline's resources and no more. It should also be possible to allocate each resource independent of other resources. For example, an in-network caching module may need large amounts of stateful memory [60] for its caches, but a routing module may need significant TCAM for routing tables.

3. **Performance isolation.** Each module should stay within its allotted ingress packets per second and bits per second rates. One module's behavior should not affect the throughput and latency of another module.

4. **Lightweight.** The isolation mechanisms themselves must have low overhead so that their presence does not significantly degrade the high performance of the underlying network device. In addition, the extra hardware consumed by these mechanisms must be small.

5. **Rapid reconfiguration.** If a module is reconfigured with new packet-processing logic, the reconfiguration process should be quick.

6. **No disruption.** If a module is reconfigured, it must not disrupt the behavior of other unchanged modules—especially important in a multi-tenant environment [40].

2.2 Target setting for Menshen

We target both programmable switches and NICs with a programmable packet-processing pipeline based on the RMT pipeline [36], a common architecture for packet processing for the highest end devices. Other projects have looked at isolation for software switches, multicore network processors, FPGA-based devices, and the Barefoot Tofino switch (without hardware changes). §6 compares against them.

An RMT pipeline can be implemented either on an FPGA (e.g., FlowBlaze [71], Lightning NIC [57], nanoPU [56]) or an ASIC (e.g., the Tofino [26], Spectrum [15], and Trident [6] switches; and the Pensando NIC [13]). This pipeline has also been embedded within larger hardware designs (e.g., PANIC [67]). Menshen builds on a baseline RMT pipeline to provide isolation between different modules/tenants. A high-speed implementation of Menshen would likely be based on an RMT ASIC. For this paper, we prototype RMT on 2 FPGA-based platforms: the NetFPGA switch [84] and the Corundum NIC [45]. Our ASIC synthesis results suggest that our lessons generalize to ASICs as well (§5.2).

3 Design

In order to meet its performance goals, RMT's pipelined architecture ensures that processing stages never stall, i.e., they can process a packet every clock cycle. The Menshen design aims to preserve this invariant so that isolation does not come at the cost of performance. To maintain this invariant, Menshen's isolation mechanisms cannot reconfigure stages or change table contents between packets. As a result,

Applied Mechanism	Targeted Resources
Space partitioning	Match action table entries, stateful memories
Overlays	Parsing actions, key extractors, packet header vector (PHV) containers, arithmetic logic units (ALUs)

Table 1: *Summary of Menshen's mechanisms.*

Menshen provides isolation by spatially partitioning switch resources between packet processing modules.

While spatial partitioning is easy for resources, e.g., match-action tables and stateful memory, that are provisioned so they can be allocated at flow granularity, it is much more challenging for resources such as key extractors (§3.1) which are generally shared across flows. This is because naive approaches to spatially partitioning such shared resources across packet-processing modules would severely reduce the number of resources available to each packet processing module—and hence the richness of that module.

To see why, consider a case where a key extractor is split between two packet processing modules: in this setting each packet processing module can only use half the key extractor, limiting its key length to half of what it would be able to use were it running on the entire pipeline. This problem is of course further exacerbated as we increase the number of packet processing modules sharing the pipeline.

Menshen addresses this problem using *overlays*: we associate a configuration lookup table with each shared resource in the switch. This lookup table is keyed by the packet processing module's ID and contains the configuration that should be used when processing packets for this module. For example, in the case of the key extractor, the configuration table contains the instructions that the module uses to construct key (§3.1). Our use of overlays means that we do not need to partition resources including ALUs or PHVs between modules. Instead, the module has exclusive access to all PHVs/ALUs in a stage when processing a packet. Table 1 summarizes our mechanisms.

To realize Menshen, on the software side, we modify an RMT compiler to target a block of resources rather than the entire pipeline. Overlays require new hardware primitives to be added to the RMT pipeline. These hardware primitives are small tables that contain per-module configurations of shared resources. On every packet, these tables are indexed using the packet's module ID to determine the configuration to use for that packet at that resource. An incremental deployment pathway for Menshen would be to only modify an RMT compiler (e.g., Tofino's compiler) to implement space partitioning without investing in new overlay hardware.

3.1 Menshen hardware

The Menshen hardware design (Figure 2) builds on RMT by adding hardware primitives for isolation into the RMT pipeline. Because these isolation primitives are added
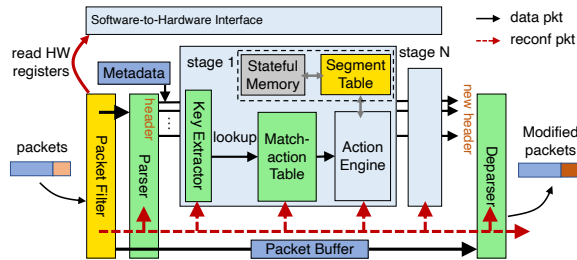
Figure 2: *Menshen hardware and software-to-hardware interface. Menshen builds on a RMT [36] pipeline, by adding Yellow components and modifying Green ones.*

pervasively throughout the pipeline, we first describe the overall Menshen hardware design including both RMT and the new isolation primitives. We then summarize the new isolation primitives added by Menshen.

Menshen expects that a data packet's header carries information identifying what module should process the packet. Currently in our prototype, this is the VLAN ID (VID) header, which we assume is set by the vSwitch [51], but other fields, e.g., VxLAN ID, can be used instead. Packets entering Menshen are first handled by a packet filter that discards packets without a VLAN ID.[2] Next, a parser extracts the VLAN ID from the packet and applies module-specific parsing to extract module-specific headers from the TCP/UDP payload. The parser then pushes these parsed packet headers into packet header vector (PHV) containers that travel through the pipeline of match-action stages.

Each stage forms keys out of headers, looks up the keys in a match-action table, and performs actions. At the start of each stage, a key extractor in the stage forms a key by combining together the headers in a module-specific manner. The keys are then concatenated with the module ID and looked up in a match-action table, whose space is partitioned across different modules. If the key matches against a match-action pair in the table, the lookup result is used to index an action table.

Similar to the match-action table, the action table is also partitioned across modules. Each action in the table identifies opcodes, operands, and immediate constants for a very-large instruction word (VLIW), controlling many parallel arithmetic and logic units (ALUs). The VLIW instruction consumes the current PHV to produce a new PHV as input for the next stage. The table's action can modify persistent pipeline state, stored in stateful memory. Stateful memory is indexed by a physical address that is computed from a local address, obtained from a module's packets. This computation is done by a segment table, which stores the offset and range of each module's slice of stateful memory. We now detail the main components of our design.

Parser. The Menshen parser is driven by a table lookup process similar to the RMT parser [36, 49]. Specifically, whenever a new packet comes in, the module ID is extracted

---
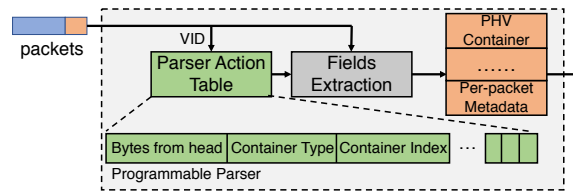
Figure 3: *Menshen programmable parser.*
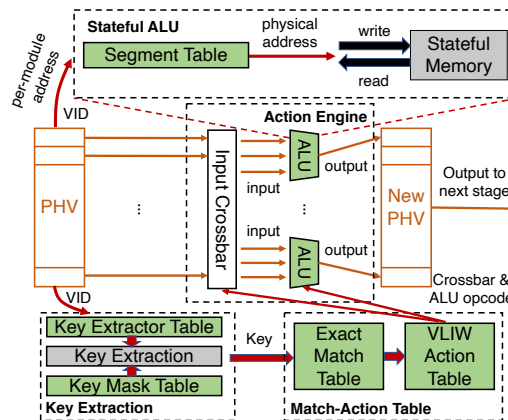
Figure 4: *Menshen processing stage.*

from its VLAN ID prior to parsing the rest of the packet. This module ID is then used as an index into the table that determines how to parse the rest of the packet (Figure 3). Each table entry corresponds to multiple parsing actions for a module—one action per extracted PHV container. Each parsing action specifies (1) *bytes from head*, indicating where in the packet the parser should extract a particular header, (2) *container type* (e.g., 4-byte container, etc.), indicating how many bytes we should extract; (3) *container index*, indicating where in the PHV we should put the extracted header into. The parser also sets aside space in the PHV for metadata that is automatically created by the pipeline (e.g., time of enqueue into switch output queues and queueing delay after dequeue) and for temporary packet headers used for computation.

Key extractor. Before a stage performs a lookup on a match-action table, a lookup key must be constructed by extracting and combining together one or more PHV containers. This key extraction process differs between modules in the same stage, and between different stages for the same module. To implement key extraction, just like the parser, we use a key extractor table (Figure 4) that is indexed by a packet's module ID. Each entry in this table specifies which PHV containers to combine together to form the key. These PHV containers are then selected into the key using a multiplexer for each portion of the key. To enable variable-length key matching for different modules, the key extractor also includes a key mask table, which also uses the module ID as an index to determine how many bits to pad in the key to bring it up to a certain fixed key size before lookup.

Match table. Each stage looks up the fixed-size key con-

---

structed by the key extractor in a match table. Currently, we support only exact-match lookup. The match table is statically partitioned across modules by giving a certain number of entries to each module. To enforce isolation among different modules, the module ID is appended to the key output by the key extractor. This augmented key is what is actually looked up against the entries in the match table; each entry stores both a key and the module ID that the key belongs to. The lookup result is used as index into the VLIW action table to identify a corresponding action to execute.

Action table and action engine. Each VLIW action table entry indicates which fields from the PHV to use as ALU operands (i.e., the configuration of each ALU's operand crossbar) and what opcode should be used for each ALU controlled by the VLIW instruction (i.e., addition, subtraction, etc.). Each ALU outputs a value based on its operands and opcode. There is one ALU per PHV container, removing the need for a crossbar on the output because each ALU's output is directly connected to its corresponding PHV container. After a stage's ALUs have modified its PHV, the modified PHV is passed to the next stage.

Stateful memory. Menshen's action engines can also modify persistent pipeline state on every packet. Each module is assigned its own address space, and the available stateful memory in Menshen is partitioned across modules. When a module accesses its slice of stateful memory, it supplies a per-module address that is translated into a physical address by a segment table before accessing the stateful memory. To perform this translation, Menshen stores per-module configuration (i.e., base address and range) in a segment table, which can be indexed by the packet's module ID. Menshen borrows this idea of a segment table from NetVRM's [79, 83] page table, but implements it in hardware instead of programming it in P4 atop Tofino's stateful memory like NetVRM does. This allows Menshen to avoid using scarce Tofino stateful memory to emulate a segment table. Also, by adding segment table hardware to each stage, Menshen avoids sacrificing the first stage of stateful memory for a segment table, instead reclaiming it for useful packet processing. This is unlike NetVRM, which can share stateful memory across modules only from the second stage because the first stage is used for the page table.

Deparser. The deparser performs the inverse operation of the parser. It takes PHV containers and writes them back into the appropriate byte offset in the packet header, merges the packet header with the corresponding payload in the packet buffer, and transmits the merged packet out of the pipeline. The format of the deparser table is identical to the parser table and is similarly indexed by a module ID.

Secure reconfiguration. Our threat model assumes that the Menshen hardware and software are trusted, but that data packets that enter the Menshen pipeline are untrusted. Data packets are untrusted because for a switch, they can come from physical machines outside the switch's control and,

for a NIC, they can come from tenant VMs sharing the NIC. Hence, the pipeline should be reconfigured only by Menshen software, not data packets.

This is a security concern faced by existing RMT pipelines as well, even without isolation support. Commercial programmable switches solve this problem by using a separate daisy chain [7] to configure pipeline stages. This chain carries configuration commands that are picked up by the intended pipeline stage as the command passes that stage. The chain is only accessible over PCIe, which is connected to the control-plane CPU, but not by Ethernet ports, which carry outside data packets. Hence, the only way to *write* new configurations into the pipeline is through PCIe. The packet-processing pipeline is restricted to just *reading* configurations and using them to implement packet processing. Thus, the daisy chain provides secure reconfiguration by physically separating reconfiguration and packet processing.

Menshen uses a similar approach by employing a daisy chain for reconfiguration when a module is updated. A special *reconfiguration packet* carries configuration commands for the pipeline's resources (e.g., parser). Our implementation of this daisy chain varies depending on the platform. For our NetFPGA prototype, this daisy chain is connected solely to the switch CPU via PCIe, similar to current switches. For our Corundum NIC prototype, we connect the daisy chain directly to PCIe and use a *packet filter* before our parser to filter out reconfiguration packets from untrusted data packets by ensuring that reconfiguration packets have a specific UDP destination port. An ideal solution would be to use a physically separate interface, e.g., USB or JTAG, for reconfiguring the Menshen pipeline on Corundum, but we found it challenging to implement such a physically separate reconfiguration interface on Corundum. In Appendix A, we show how a daisy chain permits more rapid reconfiguration than an alternative approach of using the AXI-L protocol on an FPGA.

Summary of Menshen's new primitives. The hardware primitives introduced by Menshen on top of an RMT pipeline (Figure 2) are the configuration tables for the parser, deparser, key extractor, key mask units and segment table. These tables provide an overlay feature to share the same unit across multiple modules. Specifically, for each unit, Menshen provides a table with a configuration entry per module, rather than one configuration for the whole unit. In addition, Menshen introduces the packet filter to ensure secure reconfiguration. Menshen also modifies match tables, by appending the module ID to the match key and the match-action entries. Finally, Menshen partitions match-action tables and stateful memory across all modules. These primitives ensure that updating one module only affects a single entry (for Menshen resources that use overlays) and only affects a subset of memory (for Menshen resources that use space partitioning), thus allowing us to update one module without disrupting others (§2.1).

ASIC feasibility of Menshen's primitives. Menshen's parser, deparser, key extractor, key mask, and segment tables are
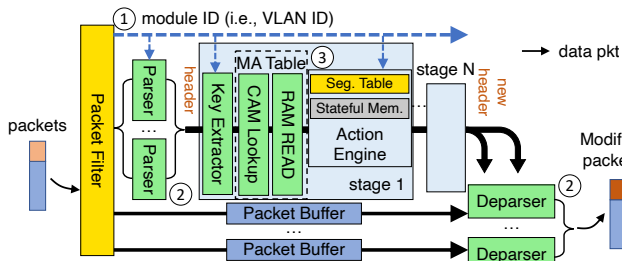
Figure 5: *Three optimization techniques applied in Menshen. Numbered circles refer to specific techniques in §3.2.*

Figure 6: *Menshen software and system-level module.*

small and simple arrays indexed by the module identifier. They can be readily realized in SRAM that can support a memory read every clock cycle. The packet filter is a simple combinational circuit that checks if the incoming packet is destined to a specific UDP destination port. Extending the match-action tables in each stage to append a module ID to every entry amounts to modestly increasing the key width in the table. While these new primitives add some additional latency relative to RMT, e.g., to go through the packet filter or reading out the per-module parser configuration, the pipelined nature of RMT means that this additional latency does not impact packet-forwarding rate. The ASIC area overhead increases as we increase the number of simultaneous programming modules that need to be supported; we quantify it in §5.2.

3.2 Improving Menshen's throughput

As shown in Figure 5, we apply 3 main techniques to optimize the forwarding performance of Menshen: (1) masking RAM read latency, (2) using multiple parsers and deparsers, and (3) increasing pipeline depth. We demonstrate the effect these techniques have on Menshen's throughput in §5.2.

① *Masking RAM read latency.* The design described in §3.1 attaches the module ID to the PHV that is sent from one element (e.g., parser, key extractor) to the next. In this design, we read the module's configuration from SRAM after the PHV arrives, thus incurring a few additional clock cycles of latency. To optimize this, we mask SRAM access latency by splitting the module ID from the PHV and sending the module ID to the next element *ahead of time*. The PHV follows the module ID, and thus the module configuration at a stage can be read concurrently with the PHV being transmitted to that stage.

② *Multiple parsers and deparsers.* In §3.1's design, there is one parser, deparser, and packet buffer. The parser extracts and parses the header and puts the full packet in the packet buffer. Then the deparser takes the modified headers from the last stage, uses them to overwrite the relevant portions of the full packet in the packet buffer, and sends out the packet.

Our optimized design uses multiple parallel parsers, deparsers, and packet buffers to improve throughput. Deparsing is the most expensive operation as any position within the PHV container might be modified, and thus any part of the packet header (128 bytes in our implementation) might need
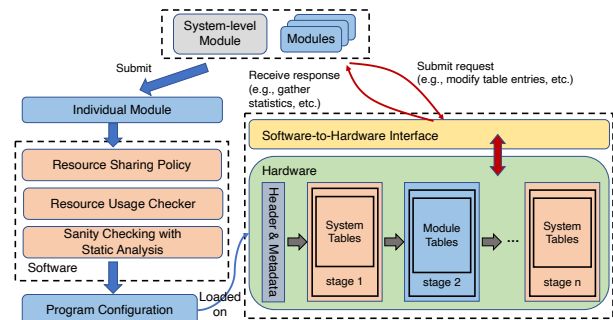
to be updated. Furthermore, deparsing has to process both the packet header and the payload. Therefore, we use 4 parallel deparsers and 2 parsers. We also associate a separate packet buffer with each deparser.

On ingress, the packet filter tags each packet with a packet buffer number (0–3) in round robin order. It also round robins incoming packets to the 2 parsers. The last pipeline stage uses the packet buffer tag to determine which packet buffer's packet the last stage's modified PHV should be combined with. Each packet buffer's deparser combines the earliest packet from the packet buffer along with the last stage's most recently modified PHV for that buffer.

③ *Deep pipelining.* With careful digital design, in Menshen's implementation, we can pipeline each element (e.g., match-action table) into several sub elements to improve throughput. For example in Figure 5, we divide the match-action table into CAM-lookup and action-RAM-read sub elements. In this specific example, this allows us to process a PHV every 2 clock cycles at each sub-element rather than every 4 clock cycles at the whole match-action table.

3.3 The Menshen system-level module

To hide information about the underlying physical infrastructure (e.g., topology) from tenant modules in a virtualized environment, modules in Menshen can use virtual IP addresses to operate in a shared environment [51]. Here, virtual IP addresses are local and scoped to modules belonging to a particular tenant, regardless of which physical device these modules are on. To support virtual IPs and provide basic services to other modules, Menshen contains a system-level module written in P4-16 that provides common OS-like functionality, e.g., converting virtual IPs to physical IPs, multicast, and looking up physical IPs to find output ports. The system-level module has 3 benefits: (1) it avoids duplication among different modules re-implementing common functions, improving the resource efficiency of the pipeline, (2) it hides underlying physical details (e.g., topology) from each module so that one tenant's modules on different network devices can form a virtual network [51], and (3) it provides common and useful real-time statistics (e.g., link utilization, queue length, etc.) that can inform packet processing within modules.

Figure 6 shows how the system-level module is laid out

relative to the other modules. Packets entering the Menshen pipeline are first processed by the system-level module before being handed off to their respective module for module-specific processing. After module-specific processing, these packets enter the system module for a second time before exiting the pipeline. The first time they enter the system-level module, packets can read and update system-level state (e.g., link utilization, packet counters, queue measurements), whereas the second time they enter the system-level module, module-specific packet header fields (e.g., virtual IP address) can be read by the system-level module to determine device-specific information (e.g., output port). In both halves, there is a narrow interface by which modules communicate with the system-level module. This split structure of the system-level module arises directly from the feed-forward nature of the RMT pipeline, where packets typically only flow forward, but not backward. Hence, packets pick up information from the system-level module in the first stage and pass information to the system-level module in the last stage. The non-system modules are sandwiched in between these two halves.

3.4 Menshen software

The software-hardware interface. The Menshen software-to-hardware interface works similar to P4Runtime [19] to support interactions (e.g., modifying match-action entries, fetching hardware statistics, etc.) between the Menshen software and the Menshen hardware. However, in addition to P4Runtime's functions, Menshen's software-hardware interface can also be used to reconfigure different hardware resources (Appendix C) in Menshen to reprogram them when a module is added or updated. This allows us to dynamically reconfigure portions of Menshen as module logic changes.

The Menshen resource checker. The Menshen resource checker ensures that each module's resource allocation complies with an operator specified resource sharing policy (e.g., dominant resource sharing (DRF) [48], or a utility-based [54] policy). In our current design we check allocations statically because reassigning resources from one module to another disrupts processing for both modules. Instead we rely on admission control and do not load a module whose resource requirements cannot be met. We leave the question of what is an appropriate resource allocation policy to future work.

The Menshen static checker. To ensure isolation, Menshen's static checker analyzes 3 properties of the module's P4 source code. First, it checks that modules do not modify hardware-related statistics (e.g., link utilization) provided by the system-level module to all modules. Second, modules can not modify their VID. This is because a module can be spread across multiple programmable devices [46, 59], and changes to VIDs by module *A* on a device can unintentionally affect a module *B* on a downstream device, where *B*'s real VID happens to be the same as *A*'s modified VID. Third, modules must not recirculate packets and their routing tables

should be loop-free.[3] This is because all modules share the same ingress pipeline bandwidth. Recirculating packets or looping them back through multiple devices will degrade the ability of other modules to process packets.

The Menshen compiler. Packet-processing pipelines (e.g., RMT [36]) are structured as feed-forward pipelines of programmable units, each of which has limited processing capabilities. This design ensures the *all-or-nothing* property: once a module has been compiled and loaded it can run at up to line rate, while modules that can not run at line rate cannot be compiled. Menshen's compiler follows the same design, and only admits modules that meet line-rate requirements.

The compiler reuses the frontend and midend of the open-source P4-16 reference compiler [18] and creates a new backend similar to BMv2 [4]. This backend has a parser, a single processing pipeline, and a deparser. The compiler takes a module's P4-16 program as input and conducts all the resource usage and static checks described above. Then, for the parser and deparser, it transforms the parser defined in the module to configuration entries for the parser and deparser tables. For the packet-processing pipeline, which consists of match-action tables, it transforms the key in a table to a configuration in the key extractor table, and actions to VLIW action table entries according to the opcodes. The compiler also performs dependency checking [36, 61] to guarantee that all ALU actions and key matches are placed in the proper stage, respecting table dependencies.

The Menshen compiler can be extended to support the same packet flowing through different P4 modules belonging to one tenant. The compiler can take multiple P4 modules as input, assign them the same module ID, and allocate them to non-overlapping pipeline stages—similar to how we lay out user and system modules in different stages as in Figure 6.

3.5 Limitations

As a research prototype, Menshen has several limitations. First, while we have developed *mechanisms* to support isolation across multiple modules, we have not yet designed *policies* that decide how much of each resource a module should be given [35]. Second, our FPGA implementation of RMT lacks many features present in a commercial RMT implementation such as the Barefoot Tofino switch [26]. Third, our compiler currently does not perform any compiler optimizations for code generation [47] or memory allocation [46, 61]. Fourth, Menshen proposes isolation mechanisms for the packet-processing pipeline, but does not deal with isolating traffic from different modules competing for output link bandwidth, which is a orthogonal traffic management problem. Proposals like PIFO [75] can be used here, by assigning PIFO ranks to different modules to realize a desired inter-module bandwidth-sharing policy.

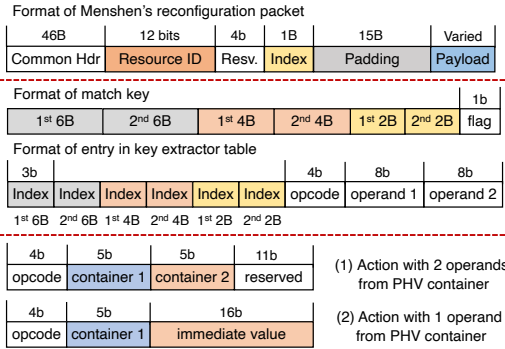[3]We check loop freedom in the control plane.

Figure 7: *Formats of Menshen's packets and tables.*

Operation	Description
add/sub	Add/subtract between containers
addi/subi	Add/subtract an immediate to/from container
set	Set a container to an immediate value
load	Load a value from stateful memory
store	Store a value to stateful memory
loadd	Load value from stateful memory, add 1, and store back
port	Set destination port
discard	Discard packet

Table 2: *Supported operations in Menshen's ALU.*

4 Implementation

4.1 Menshen hardware

To implement Menshen, we first built a baseline RMT implementation for an FPGA. Menshen includes (1) a packet filter to filter out reconfiguration packets from data packets using a specific predefined UDP destination port (i.e., 0xf1f2), (2) a programmable parser, (3) a programmable RMT pipeline with 5 programmable processing stages, (4) a deparser, and (5) a separate daisy-chain pipeline for reconfiguration. It also includes Menshen's primitives for isolation. We have integrated it into both the Corundum NIC [45] and the NetFPGA reference switch [84]. The Menshen code base together with the optimizations (§3.2) consists of 9975 lines of Verilog. Of this, 3098 and 3226 lines are for handling data bus widths of 512 bits (Corundum) and 256 bits (NetFPGA) respectively. 3651 lines are for the common blocks, e.g., key extractor, etc. Below, we describe our hardware implementation in more detail. Figure 7 shows the formats of Menshen's packets and tables.

PHV format. Menshen's PHV has 3 types of containers of different sizes, namely 2-byte, 4-byte and 6-byte containers. Each type has 8 containers. Also, we allocate and append an additional 32 bytes to store platform-specific metadata (e.g., an indication to drop the packet, destination port, etc.), which results in a PHV length of 128 bytes in total. Thus, we have a total of $3 * 8 + 1 = 25$ PHV containers. To prevent any possibility of PHV contents leaking from one module to another, the PHV is zeroed out for each incoming packet.

Reconfiguration packet format. Figure 7 shows the format of Menshen reconfiguration packets. The reconfiguration packet is a UDP packet with the standard UDP, Ethernet, VLAN, and IP headers. Within the UDP payload, a 12-bit resource ID indicates which hardware resource within which stage should be updated (e.g., key extractor table in stage 3). To reconfigure the resource, the table storing the configuration for this resource must be updated by writing the entry stored within the reconfiguration packet's payload at the location specified by the 1-byte index field in the reconfiguration packet header. The UDP destination port field determines whether the reconfiguration packet is valid or not.

Packet filter. The packet filter has 2 registers that can be accessed by the Menshen software via Xilinx's AXI-Lite protocol [28]: (1) a 4-byte reconfiguration packet counter, which monitors how many reconfiguration packets have passed through the daisy chain; (2) a 32-bit bitmap, which indicates which module is currently being updated (e.g., bit 1 stands for module 1, bit 2 for module 2, etc.). During reconfiguration of a module, via the software-to-hardware interface, the Menshen software reads the reconfiguration packet counter. It then writes the bitmap to reflect the module ID M of the module currently being updated. The bitmap is then consulted on every packet to drop data packets from M until reconfiguration completes, so that M's "in-flight" packets aren't incorrectly processed by partial configurations.

Then, the Menshen software sends all reconfiguration packets embedded with the predefined UDP destination port to the daisy chain. Finally, it polls the reconfiguration packet counter to check if reconfiguration is over and then zeroes the bitmap so that M's packets are no longer dropped. Reconfiguration packets maybe dropped before they reach the RMT pipeline. This can be detected by polling the reconfiguration packet counter to see if it has been correctly incremented or not. If it hasn't been incremented correctly, then the entire reconfiguration process restarts with M's packets being dropped until reconfiguration is successful.

Programmable parser/deparser. We currently support per-module packet header parsing in the first 128 bytes of the packet. These 128 bytes also include the headers common to all modules (e.g., Ethernet, VLAN, IP, and UDP). We design the parser action for each parsed PHV container as a 16-bit action. The first 3 bits are reserved. The next 7 bits indicate the starting extraction position in bytes from byte 0. These 7 bits can cover the whole 128-byte length. Then, the next 2 bits and 3 bits indicate the container type (2, 4, or 6 byte) and number (0–7) respectively. The last bit is the validity bit. For each module, we allocate 10 such parser actions (i.e., to parse out at most 10 containers), resulting in a 160-bit-wide entry for the parser action table.

We note that we only parse out fields of a packet into PHV containers, if those fields are actually used as part of either keys or actions in match-action tables. Before packets are sent out, the deparser pulls out the full packet (including the payload) from the packet buffer and only updates the portions of the packet that were actually modified by table actions. This approach allows us to reduce the number of PHV containers to

25 because packet fields that are never modified or looked up by the Menshen pipeline need not travel along with the PHV.

Key extractor. The key for lookup in the match-action table is formed by concatenating together up to 2 PHV containers each of the 2-byte, 4-byte, and 6-byte container types. Hence the key can be up to 24 bytes and 6 containers long. Since there are 8 containers per type, the key extraction table entry for each module in each stage uses $log_2(8) * 6 = 18$ bits to determine which container to use for the 6 key locations. Additionally, the key extractor is also used to support conditional execution of actions based on the truth value of a predicate of the form *A OP B*, where *A* and *B* are packet fields and *OP* is a comparison operator. For this purpose, each key extractor table entry also specifies the 2 operands for the comparison operation and the comparison opcode. The opcode is a 4-bit number, while the operands are 8 bits each. The operands can either be an immediate value or refer to one of the PHV containers. The result of the predicate evaluation adds one bit to the original 24 byte key, bringing the total key length to $24 * 8 + 1 = 193$ bits. Because not all keys need to be 193 bits long, we use a 193-bit-wide mask table. Each entry in this table denotes the validity of each of the 193 key bits for each module in each stage. This is somewhat wasteful and can be improved by storing validity information within the key extractor table itself.

Exact match table. To implement the exact match table, we leverage the Xilinx CAM block [31]. This CAM matches the key from the key extractor module against the entries within the CAM. As discussed in §3.1, to ensure isolation between different modules, we append the module ID (i.e., VLAN ID) to each entry, which means that the CAM has a width of $193 + 12 = 205$ bits. The lookup result from the CAM is used to index the VLIW action table. The action is designed in a 25-bit format per ALU/container (Figure 7). As we have $24 + 1 = 25$ PHV containers, the width of the VLIW action table is $25 * 25 = 625$ bits. The Xilinx CAM block simplifies implementation of an exact-match table and can also easily support ternary matches if needed (Appendix B).

Action engine. The crossbar and ALUs in the action engine use the VLIW actions to generate inputs for each ALU and carry out per-ALU operations. ALUs support simple arithmetic, stateful memory operations (e.g., loads and stores), and platform-specific operations (e.g., discard packets) (Table 2). The formats of these actions are shown in Figure 7. Additionally, in stateful ALU processing, each entry in the segment table is a 2-byte number, where the first byte and second byte indicate memory offset and range, respectively.

Menshen primitives. Menshen's isolation primitives (e.g., key-extractor and segment tables) are simple arrays implemented using the Xilinx Block RAM [30] feature.

4.2 Menshen Software

The Menshen compiler reuses the open-source P4-16 reference compiler [18] and implements a new backend

Program	Description
CALC [20]	return value based on parsed opcode and operands
Firewall [20]	stateless firewall that blocks certain traffic
Load Balancing [20]	steer traffic based on 4-tuple header info
QoS [20]	set QoS based on traffic type
Source Routing [20]	route packets based on parsed header info
NetCache [60]	in-network key-value store
NetChain [59]	in-network sequencer
Multicast [20]	multicast based on destination IP address

Table 3: *Evaluated use cases.*

extension in 3773 lines of C++. It takes the module written in P4-16 together with resource allocation as the inputs, and generates per-module configurations for Menshen hardware. Specifically, it (1) conducts resource usage checking to ensure every program's resource usage is below its allocated amount; (2) places the system-level module's (120 lines of P4-16) configurations in the first and last stages in the Menshen pipeline; and (3) allocates PHV containers to the fields shared between the system-level and other modules so that the other modules can be sandwiched between the two halves of the system-level module (§3.4). The Menshen software-to-hardware interface is written in Python. It configures Menshen hardware by converting program configurations to reconfiguration packets.

4.3 Corundum and NetFPGA integrations

We have integrated Menshen into 2 FPGA platforms: one for the NetFPGA platform that captures the hardware architecture of a switch [84], and another for the Corundum platform that captures the hardware architecture of a NIC [45]. Menshen's integration on Corundum [45] is based on a 512-bit AXI-S [29] data width and runs at 250 MHz. Although Menshen's pipeline can be integrated into both the sending and receiving path, in our current implementation, we have integrated Menshen into only Corundum's sending path, i.e., PCIe input to Ethernet output. Menshen on NetFPGA [84] uses a 256-bit AXI-S [29] data width and runs at 156.25 MHz.

On the Corundum NIC platform, we insert a 1-bit discard flag, while on the NetFPGA switch platform, we insert a 1-bit discard flag and 128-bit platform-specific metadata, i.e., source port, destination port and packet length, into the PHV's metadata field. A 4-bit one-hot encoded tag indicates the packet buffer (§3.2). The table depth in Menshen's parser, key extractor, key mask, page, and deparser tables affects the maximum number of modules we can support and is currently 32. The depth of CAM and VLIW action table directly influences the amount of match-action entries and VLIW actions that can be allocated to all modules. Due to the open technical challenge of implementing CAMs on FPGAs efficiently [58,71], we set their depth to 16 in each stage. While 16 is a small depth, the depth can be improved by using a hash table, rather than a CAM, for exact matching, e.g., cuckoo hashing [69].

5 Evaluation

In §5.1, we show that Menshen can meet our requirements (§2.1): it can be rapidly reconfigured, is lightweight, provides behavior isolation, and is disruption-free. Menshen achieves

performance isolation by (1) assuming packets exceed a minimum size (to guarantee line rate) and (2) forbidding recirculation. If either is violated, hardware rate limiters can be used to limit each module's packet/bit rate. It achieves resource isolation by ensuring that a table entry for a resource (e.g., parser) is allotted to at most one module. In §5.2, we evaluate the current performance of Menshen.

Experimental setup. To demonstrate Menshen's ability to provide multi-module support, we picked 6 tutorial P4 programs [20], as detailed in Table 3, together with simplified versions of NetCache [60] and NetChain [59].[4] The system-level module provides basic forwarding and routing, with multicast logic integrated in it. Menshen's parameters are detailed in §4 and summarized in Table 5 in the Appendix.

Testbed. We evaluate Menshen based on our Corundum and NetFPGA integrations as described in §4. For the switch platform experiments on NetFPGA, we use a single quad-port NetFPGA SUME board [14], where two ports are connected to a machine equipped with an Intel Xeon E5645 CPU clocked at 2.40 GHz and a dual-port Intel XXV710 10/25GbE NIC. For the NIC platform experiments on Corundum, we use a single Xilinx Alveo U250 board [2], where one port is with Menshen for the transmitting path and this port is connected to a 100 GbE NIC as the receiving path. Both setups are used to check Menshen's correctness (§5.1). For NetFPGA performance tests (§5.2), we use the host as a packet generator. For Corundum performance tests (§5.2), we internally connect its receiving and transmitting path, and use the Spirent tester [22] to generate traffic. We depict our testing setup in Appendix D.

5.1 Does Menshen meet its requirements?

Menshen can be rapidly reconfigured. Reconfiguration time includes both the software's compilation time (Figure 8) and the hardware's configuration time (Figure 9); we evaluate each separately. When a module is compiled, the compiler needs to generate both configuration bits for various hardware resources as well as match-action entries for the tables the module looks up. These match-action entries can and will be overwritten by the control plane, but we need to start out with a new set of match-action entries for a module to ensure no information leaks from a previous module.

Hence, every time a module is compiled, the compiler also generates match-action entries. Within an exact match table, these entries must be different from each other to prevent multiple lookup results. As a result, Menshen's compilation time increases with the number of match-action entries in the module (Figure 8). To contextualize this, Menshen's compile times (few seconds) compare favorably to compile times for Tofino (~10 seconds for our use cases) and FPGA synthesis times (10s of minutes). We note that this is an imperfect comparison: our compiler performs fewer optimizations than

Hardware Implementation	Slice LUTs	Block RAMs
NetFPGA reference switch	42325 (9.77%)	245.5 (16.7%)
RMT on NetFPGA	200573 (46.3%)	641 (43.6%)
Menshen on NetFPGA	200733 (46.34%)	641 (43.6%)
Corundum	61463 (3.56%)	349 (12.98%)
RMT on Corundum	235686 (13.63%)	316 (11.75%)
Menshen on Corundum	235903 (13.65%)	316 (11.75%)

Table 4: *Resources used by 5-stage Menshen pipeline, on NetFPGA SUME and AU250 boards, compared with reference switch, Corundum NIC, and RMT.*

either the Tofino or FPGA compilers and our targets are simpler. That said, compilation can happen offline, and hence it is not as time-sensitive compared to run-time reconfiguration.

To measure time taken for Menshen's configuration post compilation, we vary the number of entries the Menshen software has to write into the pipeline.[5] Also, as a comparison, we evaluate the cost of the Tofino run-time APIs from Tofino SDE 9.0.0 to insert match-action table entries for the CALC program. From Figure 9, we observe that the time spent in configuration of the hardware via Menshen's software-to-hardware interface is similar to Tofino's run-time APIs.

Menshen can reconfigure without disruption. To show Menshen can support disruption-free reconfiguration, we launch three CALC programs with fixed input packet rate, i.e., 5:3:2 ratio on a single link for module 1, 2 and 3, respectively. We use netmap-based tcpreplay to generate total traffic of 9.3 Gbit/s on a 10 Gbit/s link. 0.5 seconds in, we start to reconfigure the first module to see if the packet processing of other modules has stalled or not. In Figure 10 we show the throughput achieved by each of three modules when reconfiguring module 1. We can observe that model 2 and 3 see no impact on their throughput. This demonstrates that Menshen provides performance isolation, and that it is feasible for a tenant to reconfigure their module without impacting other tenants. By contrast, updating a module on Tofino (§6) requires resetting the entire switch pipeline. Even with Tofino's Fast Refresh [9], this leads to a 50 ms disruption of all servers (and their VMs) whose traffic is routed through the switch. This disruption can be significant in public cloud environments, and in many cases renders dynamic reconfiguration infeasible.

Menshen is lightweight. We list Menshen's resource usage of logic and memory (i.e., LUTs and Block RAMs), including absolute numbers and fractions, in Table 4. For comparison, we also list the resource usage of the NetFPGA reference switch and the Corundum NIC. We believe that the additional hardware footprint of Menshen is acceptable for the programmability and isolation mechanisms it provides relative to the base platforms. The reason that Menshen uses more LUTs than Block RAMs is that Menshen leverages the Shift Register Lookup (SRL)-based implementation of Xilinx's CAM IP [31]. We also compared with an RMT design, where we modified Menshen's hardware to support only one module. Relative to RMT, Menshen incurs an extra

---

[4]Our versions of NetChain and NetCache do not include some features such as tagging hot keys.

[5]Since the Menshen hardware can't currently support so many entries (§4.3), we overwrite previously written entries to measure configuration time.
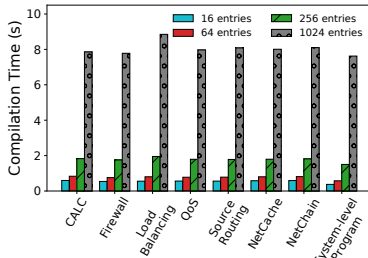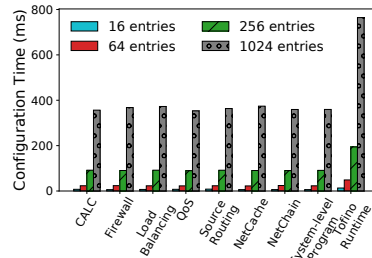
Figure 8: *Compilation time.*
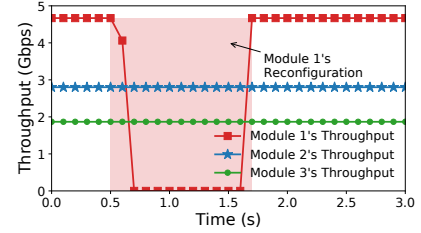

Figure 9: *Configuration time.*

Figure 10: *Throughput during reconfiguration.*

0.65% (NetFPGA) and 0.15% (Corundum) in LUTs usage.

Menshen provides behavior isolation. Next, we spot check that Menshen can correctly isolate modules, i.e., every running module can concurrently execute its desired functionality. For this, we ran the CALC, Firewall, and NetCache module simultaneously on the Menshen pipeline. We generate data packets of different VIDs, which indicate which of these 3 modules they belong to, and input them to the Menshen FPGA prototype on both platforms. By examining the output packets at the end of Menshen's pipeline, we checked that Menshen had correctly isolated the modules, i.e., each module behaved as it would have had it run by itself. We repeated the same experiment by running the Load Balancing, Source Routing, and NetChain modules simultaneously; we observed correct behavior isolation here too.

5.2 Menshen Performance

How many modules can be packed? In our current prototype on both Corundum and NetFPGA, we can support at most 32 modules because each isolation primitive (e.g., key extractor table) currently has 32 entries. In practice, the number of modules could be less than 32 if modules need to share a more bottlenecked hardware resource. For instance, if each module wants a match-action entry in every pipeline stage, the maximum number of modules is at most 16 because there are only 16 match-action entries in each stage in our current prototype. However, the numbers above are entirely a function of how much hardware one is willing to pay in exchange for multitenancy support. If we can afford to expend additional resources on an FPGA or extra area on an ASIC, we can correspondingly support a larger number of modules.

Latency. In our current implementation, the number of clock cycles needed to process a packet in the pipeline depends on packet size. This is because the number of cycles to process both the header and the payload depend on the header and payload length. For instance, for a minimum packet size of 64 bytes, Menshen's pipeline introduces 79 and 106 cycles of processing for NetFPGA and Corundum, resulting in $79 * \frac{1000}{156.25} = 505.6$ ns and $106 * \frac{1000}{250} = 424$ ns latency, respectively. For the max. packet size of 1500 bytes, Menshen incurs 146 and 112 cycles for NetFPGA and Corundum, resulting in $150 * \frac{1000}{156.25} = 960$ ns and $129 * \frac{1000}{250} = 516$ ns latency.

Throughput. For NetFPGA, we used MoonGen [42] to generate packets with different sizes. Figure 11a shows that Menshen achieves a rate of 10 Gbit/s after a packet size of 96 bytes. This is the maximum supported by our MoonGen setup because we have a single 10G NIC. For Corundum, we internally connected Corundum's receiving and transmitting path. Rather than using a host-based packet generator through PCIe, we used Spirent FX3-100GO-T2 tester to test Menshen's throughput. The MTU size is set to 1500 bytes. As shown in Figure 11b and Figure 11c, optimized Menshen on Corundum achieves 100 Gbit/s at 256 bytes, while unoptimized Menshen can only achieve 80 Gbit/s at MTU-size packets. Also, we sample packets to evaluate the packet latency of optimized Menshen on Corundum with full rate. As depicted in Figure 11d, at full rate, it incurs about 1.2 μs latency.

ASIC feasibility. With the same parameter settings in §5, we use the Synopsys DC synthesis tool [24] and FreePDK45nm technology library [8] to assess the ASIC feasibility of the Menshen pipeline.[6] At 1 GHz frequency, when compared with an RMT design, where we modified Menshen to support only one module, Menshen incurs 18.5%, 7%, 20.9% additional chip area for the parser, deparser and one stage, respectively. For a 5-stage pipeline along with the packet filter, parser, deparser and packet buffers, Menshen (10.81 mm^2) incurs 11.4% additional chip area compared with RMT (9.71 mm^2).

Considering that memory (i.e., lookup tables) and packet processing logic only costs at most 50% in switch chip area [21, page 36], Menshen's chip area overhead is moderate (11.4% * 50% = 5.7%), which is conservative since the number of entries in our match-action table is only 16 (§4.1). With much larger number of entries in lookup tables—which is the common block between Menshen and RMT—Menshen's additional chip area will be negligible.

6 Related work

Multi-core architecture solutions. To support isolation on programmable network devices based on multi-cores [10, 11, 23], FairNIC [50] partitions cores, caches, and memory across tenants and shares bandwidth across tenants through Deficit Weighted Round Robin (DWRR) scheduling.

---

[6]Since we can not have access to source code of Xilinx IPs (e.g., DMA, Ether+PHY, etc.), we solely run synthesis on Menshen's Verilog codebase.

(a) *Optimized NetFPGA.* **(b)** *Optimized Corundum.* **(c)** *Unoptimized Corundum.* **(d)** *Optimized Corundum latency.*
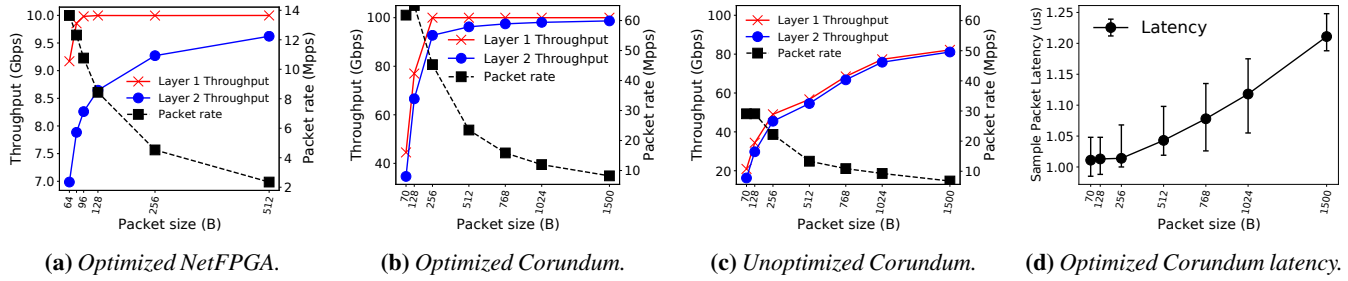
Figure 11: *Results for performance benchmarks.*

iPipe [68] uses a hybrid DRR+FCFS scheduler to share SmartNIC and host processors between different programs. Menshen uses space partitioning as well to allocate different resources to different modules. However, RMT's spatial/-dataflow architecture differs considerably from the Von Neumann architectures for multi-core network processors targeted by FairNIC and iPipe. An RMT architecture can not support a runtime system similar to the ones used by iPipe and FairNIC.

FPGA-based solutions. Several FPGA platforms exist for programmable packet processing. These platforms can be broadly categorized into (1) direct programming of FPGAs [12, 44, 55, 64, 73, 77, 78] and (2) higher-level abstractions built on top of FPGAs [33, 37, 43, 71].

Systems (e.g., VirtP4 [73], MTPSA [77]) based on direct FPGA programming typically implement packet-processing logic in a hardware-description language (HDL) or using a high-level language like P4 [55, 78] or C [32, 64] that is translated into HDL. The HDL program is fed to an FPGA synthesis tool to produce a bitstream, which is written into the FPGA. This approach requires combining the programs of different modules into a single Verilog program, which can then be fed to the synthesis tool. Thus, changing one module disrupts other modules, violating our requirement of no disruption.

FlowBlaze [71], SwitchBlade [33], and hXDP [37] expose a restricted higher-level abstraction like RMT or eBPF on top of an FPGA. FlowBlaze and hXDP do not provide support for isolation. SwitchBlade does, but its higher-level abstraction is much less flexible than the RMT abstraction in Menshen. NICA [43] targets an FPGA NIC and is designed to share one pre-programmed offloading engine across many modules, while Menshen also targets ASIC pipelines and supports reprogramming individual modules without disrupting others.

Tofino [26]. Tofino is a commercial switch ASIC that uses multiple parallel RMT pipelines. However, Tofino currently does not support multiple modules/P4 programs within a single pipeline. The current Tofino compiler requires a single P4 program per pipeline. Multiple P4 programs can be merged into a single program per pipeline and then fed into the Tofino compiler (Wang et al. [79] and μP4 [76]). However, both approaches still disrupt all tenants every time a single tenant in any pipeline is updated. This is because despite supporting an independent program per pipeline, updating any of these programs requires a reset of the entire Tofino switch [9].

Emulation-based solutions. Hyper4 [53] and HyperV [81] propose to emulate multiple P4 programs/modules using a single hypervisor P4 program, which can be configured at run time by the control plane, thus supporting disruption-free reconfiguration. However, we found that it was very challenging to design a sufficiently "universal" hypervisor program on a commercial RMT switch like Tofino.

As one example, the hypervisor program needs to support performing a bit-shift by an amount determined by a packet field, where the packet field is specified by the control plane. However, a high-speed chip like Tofino has several restrictions on bit-shifts and other computations for performance, e.g., on Tofino, the shift width and field to shift must be supplied at compile time, not at run time by the control plane.

PANIC [67] and FlexCore [80]. PANIC and FlexCore [80] are programmable multi-tenant NIC and switch designs, respectively. They both suffer from scalability issues because they need to build a large crossbar with long wires interconnecting all engines to each other, which requires careful physical design [38, Appendix C]. Menshen's RMT pipeline is easier to scale as its wires are shorter: they only connect adjacent pipeline stages [36, 2.1].

7   Conclusion

This paper described Menshen, a system for isolating co-resident packet-processing modules on pipelines similar to RMT. Menshen builds on the idea of space partitioning and overlays, and is comprised of a set of simple hardware primitives that are inserted at different points in an RMT pipeline. These primitives are straightforward to realize in both ASICs and FPGAs. Menshen thus demonstrates that providing inter-module isolation in high-speed packet-processing pipelines is practical. Our software and hardware are available at https://isolation.quest/.

References

[1] About Arm Debugger Support for Overlays. `https://developer.arm.com/documentation/101470/2021-0/Debugging-Embedded-Systems/About-Arm-Debugger-support-for-overlays?lang=en`.

[2] Alveo U250 Data Center Accelerator Card. `https://www.xilinx.com/products/boards-and-kits/alveo/u250.html`.

[3] Before Memory was Virtual. `http://160592857366.free.fr/joe/ebooks/ShareData/Before%20Memory%20was%20Virtual%20By%20Peter%20J.%20Denning%20from%20George%20Mason%20University.pdf`.

[4] Behavioral model targets. `https://github.com/p4lang/behavioral-model/blob/master/targets/README.md`.

[5] Bidirectional Forwarding Detection (BFD). `https://tools.ietf.org/html/rfc5880`.

[6] BROADCOM Trident Programmable Switch. `https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56870-series`.

[7] Daisy Chain. `https://en.wikipedia.org/wiki/Daisy_chain_(electrical_engineering)`.

[8] FreePDK45. `https://www.eda.ncsu.edu/wiki/FreePDK45:Contents`.

[9] Leveraging Stratum and Tofino Fast Refresh for Software Upgrades. `https://opennetworking.org/wp-content/uploads/2018/12/Tofino_Fast_Refresh.pdf`.

[10] LiquidIO Smart NICs. `https://www.marvell.com/products/ethernet-adapters-and-controllers/liquidio-smart-nics.html`.

[11] Mellanox BlueField VPI 100Gps SmartNIC. `https://www.mellanox.com/files/doc-2020/pb-bluefield-vpi-smart-nic.pdf`.

[12] Mellanox Innova Open Programmable SmartNIC. `https://www.mellanox.com/sites/default/files/doc-2020/pb-innova-2-flex.pdf`.

[13] Naples DSC-100 Distributed Services Card. `https://pensando.io/assets/documents/Naples_100_ProductBrief-10-2019.pdf`.

[14] NetFPGA-SUME Virtex-7 FPGA Development Board. `https://reference.digilentinc.com/reference/programmable-logic/netfpga-sume/start`.

[15] NVIDIA Mellanox Spectrum Switch. `https://www.mellanox.com/files/doc-2020/pb-spectrum-switch.pdf`.

[16] Operating Systems Three Easy Pieces. `https://iitd-plos.github.io/os/2020/ref/os-arpaci-dessau-book.pdf`.

[17] Overlaying in Commodore. `https://www.atarimagazines.com/compute/issue73/loading_and_linking.php`.

[18] P4-16 Reference Compiler. `https://github.com/p4lang/p4c`.

[19] P4 Runtime. `https://p4.org/p4-runtime/`.

[20] P4 Tutorial. `https://github.com/p4lang/tutorials`.

[21] Programmable Forwarding Planes are Here to Stay. `https://conferences.sigcomm.org/sigcomm/2017/files/program-netpl/01-mckeown.pptx`.

[22] Spirent Quint-Speed High-Speed Ethernet Test Modules. `https://assets.ctfassets.net/wcxs9ap8i19s/12bhgz12JBkRa66QUG4N0L/af328986e22b1694b95b290c93ef6c21/Spirent_fX3_HSE_Module_datasheet.pdf`.

[23] Stingray SmartNIC Adapters and IC. `https://www.broadcom.com/products/ethernet-connectivity/smartnic`.

[24] Synopsys DC Ultra. `https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html`.

[25] The Space Shuttle Fight Software Development Process. `https://www.nap.edu/read/2222/chapter/5`.

[26] Tofino: P4-programmable Ethernet switch ASIC. `https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html`.

[27] Von Neumann Architecture. `https://en.wikipedia.org/wiki/Von_Neumann_architecture`.

[28] Xilinx AXI4-Lite Interface Protocol. `https://www.xilinx.com/products/intellectual-property/axi.html`.

[29] Xilinx AXI4-Stream. `https://www.xilinx.com/products/intellectual-property/axi4-stream_interconnect.html`.

[30] Xilinx Block Memory Generator v8.4. `https://www.xilinx.com/support/documentation/ip_documentation/blk_mem_gen/v8_4/pg058-blk-mem-gen.pdf`.

[31] Xilinx Parameterizable Content-Addressable Memory. `https://www.xilinx.com/support/documentation/application_notes/xapp1151_Param_CAM.pdf`.

[32] Xilinx Vitis High-Level Synthesis. `https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html`.

[33] M. B. Anwer, M. Motiwala, M. b. Tariq, and N. Feamster. SwitchBlade: A Platform for Rapid Deployment of Network Protocols on Programmable Hardware. In *ACM SIGCOMM*, 2010.

[34] K. Arvind and R. S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE TC*, 1990.

[35] M. Blöcher, L. Wang, P. Eugster, and M. Schmidt. Switches for HIRE: Resource Scheduling for Data Center in-Network Computing. In *ACM ASPLOS*, 2021.

[36] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *ACM SIGCOMM*, 2013.

[37] M. S. Brunella, G. Belocchi, M. Bonola, S. Pontarelli, G. Siracusano, G. Bianchi, A. Cammarano, A. Palumbo, L. Petrucci, and R. Bifulco. hXDP: Efficient Software Packet Processing on FPGA NICs. In *USENIX OSDI*, 2020.

[38] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Vargaftik, A. Berger, G. Mendelson, M. Alizadeh, S.-T. Chuang, I. Keslassy, A. Orda, and T. Edsall. dRMT: Disaggregated Programmable Switching. In *ACM SIGCOMM*, 2017. Tech report available at `https://cs.nyu.edu/~anirudh/sigcomm17_drmt_extended.pdf`.

[39] J. B. Dennis and D. P. Misunas. A Preliminary Architecture for a Basic Data-Flow Processor. In *ACM ISCA*, 1974.

[40] S. Dharanipragada, S. Joyner, M. Burke, J. Nelson, I. Zhang, and D. R. K. Ports. PRISM: Rethinking the RDMA Interface for Distributed Systems, 2021.

[41] N. Dukkipati. *Rate Control Protocol (RCP): Congestion Control to Make Flows Complete Quickly*. PhD thesis, Stanford University, 2008.

[42] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle. MoonGen: A Scriptable High-Speed Packet Generator. In *ACM IMC*, 2015.

[43] H. Eran, L. Zeno, M. Tork, G. Malka, and M. Silberstein. NICA: An Infrastructure for Inline Acceleration of Network Applications. In *USENIX ATC*, 2019.

[44] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *USENIX NSDI*, 2018.

[45] A. Forencich, A. C. Snoeren, G. Porter, and G. Papen. Corundum: An Open-Source 100-Gbps NIC. In *IEEE FCCM*, 2020.

[46] J. Gao, E. Zhai, H. H. Liu, R. Miao, Y. Zhou, B. Tian, C. Sun, D. Cai, M. Zhang, and M. Yu. Lyra: A Cross-Platform Language and Compiler for Data Plane Programming on Heterogeneous ASICs. In *ACM SIGCOMM*, 2020.

[47] X. Gao, T. Kim, M. D. Wong, D. Raghunathan, A. K. Varma, P. G. Kannan, A. Sivaraman, S. Narayana, and A. Gupta. Switch Code Generation Using Program Synthesis. In *ACM SIGCOMM*, 2020.

[48] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *USENIX NSDI*, 2011.

[49] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown. Design Principles for Packet Parsers. In *ACM/IEEE ANCS*, 2013.

[50] S. Grant, A. Yelam, M. Bland, and A. C. Snoeren. SmartNIC Performance Isolation with FairNIC: Programmable Networking for the Cloud. In *ACM SIGCOMM*, 2020.

[51] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *ACM SIGCOMM*, 2009.

[52] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger. Sonata: Query-Driven Streaming Network Telemetry. In *ACM SIGCOMM*, 2018.

[53] D. Hancock and J. van der Merwe. HyPer4: Using P4 to Virtualize the Programmable Data Plane. In *ACM CoNEXT*, 2016.

[54] M. Hogan, S. Landau-Feibish, M. T. Arashloo, J. Rexford, and D. Walker. Modular Switch Programming Under Resource Constraints. In *USENIX NSDI*, 2022.

[55] S. Ibanez, G. Brebner, N. McKeown, and N. Zilberman. The P4->NetFPGA Workflow for Line-Rate Packet Processing. In *ACM/SIGDA FPGA*, 2019.

[56] S. Ibanez, A. Mallery, S. Arslan, T. Jepsen, M. Shahbaz, N. McKeown, and C. Kim. The nanoPU: Redesigning the CPU-Network Interface to Minimize RPC Tail Latency, 2020.

[57] S. Ibanez, M. Shahbaz, and N. McKeown. The Case for a Network Fast Path to the CPU. In *ACM HotNets*, 2019.

[58] W. Jiang. Scalable Ternary Content Addressable Memory Implementation Using FPGAs. In *ACM/IEEE ANCS*, 2013.

[59] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica. NetChain: Scale-Free Sub-RTT Coordination. In *USENIX NSDI*, 2018.

[60] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *ACM SOSP*, 2017.

[61] L. Jose, L. Yan, G. Varghese, and N. McKeown. Compiling Packet Programs to Reconfigurable Switches. In *USENIX NSDI*, 2015.

[62] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford. HULA: Scalable Load Balancing Using Programmable Data Planes. In *ACM SOSR*, 2016.

[63] J. Krude, J. Hofmann, M. Eichholz, K. Wehrle, A. Koch, and M. Mezini. Online Reprogrammable Multi Tenant Switches. In *1st ACM CoNEXT Workshop on Emerging In-Network Computing Paradigms*, 2019.

[64] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *ACM SIGCOMM*, 2016.

[65] Y. Li, R. Miao, C. Kim, and M. Yu. FlowRadar: A Better NetFlow for Data Centers. In *USENIX NSDI*, 2016.

[66] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh, and M. Yu. HPCC: High Precision Congestion Control. In *ACM SIGCOMM*, 2019.

[67] J. Lin, K. Patel, B. E. Stephens, A. Sivaraman, and A. Akella. PANIC: A High-Performance Programmable NIC for Multi-tenant Networks. In *USENIX OSDI*, 2020.

[68] M. Liu, T. Cui, H. Schuh, A. Krishnamurthy, S. Peter, and K. Gupta. Offloading Distributed Applications onto SmartNICs Using IPipe. In *ACM SIGCOMM*, 2019.

[69] R. Pagh and F. F. Rodler. Cuckoo Hashing. *Journal of Algorithms*, 2004.

[70] Y. Park, H. Park, and S. Mahlke. CGRA Express: Accelerating Execution Using Dynamic Operation Fusion. In *ACM CASES*, 2009.

[71] S. Pontarelli, R. Bifulco, M. Bonola, C. Cascone, M. Spaziani, V. Bruschi, D. Sanvito, G. Siracusano, A. Capone, M. Honda, F. Huici, and G. Siracusano. FlowBlaze: Stateful Packet Processing in Hardware. In *USENIX NSDI*, 2019.

[72] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. R. K. Ports, and P. Richtarik. Scaling Distributed Machine Learning with In-Network Aggregation. In *USENIX NSDI*, 2021.

[73] M. Saquetti, G. Bueno, W. Cordeiro, and J. R. Azambuja. Hard Virtualization of P4-Based Switches with VirtP4. In *ACM SIGCOMM Posters and Demos*, 2019.

[74] A. Sivaraman, C. Kim, R. Krishnamoorthy, A. Dixit, and M. Budiu. DC.P4: Programming the Forwarding Plane of a Data-Center Switch. In *ACM SOSR*, 2015.

[75] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown. Programmable Packet Scheduling at Line Rate. In *ACM SIGCOMM*, 2016.

[76] H. Soni, M. Rifai, P. Kumar, R. Doenges, and N. Foster. Composing Dataplane Programs with μP4. In *ACM SIGCOMM*, 2020.

[77] R. Stoyanov and N. Zilberman. MTPSA: Multi-Tenant Programmable Switches. In *3rd P4 Workshop in Europe*, 2020.

[78] H. Wang, R. Soulé, H. T. Dang, K. S. Lee, V. Shrivastav, N. Foster, and H. Weatherspoon. P4FPGA: A Rapid Prototyping Framework for P4. In *ACM SOSR*, 2017.

[79] T. Wang, H. Zhu, F. Ruffy, X. Jin, A. Sivaraman, D. R. K. Ports, and A. Panda. Multitenancy for Fast and Programmable Networks in the Cloud. In *USENIX HotCloud*, 2020.

[80] J. Xing, K.-F. Hsu, M. Kadosh, A. Lo, Y. Piasetzky, A. Krishnamurthy, and A. Chen. Runtime Programmable Switches. In *USENIX NSDI*, 2022.

[81] C. Zhang, J. Bi, Y. Zhou, A. B. Dogar, and J. Wu. HyperV: A High Performance Hypervisor for Virtualization of the Programmable Data Plane. In *IEEE ICCCN*, 2017.

[82] P. Zheng, T. Benson, and C. Hu. P4Visor: Lightweight Virtualization and Composition Primitives for Building and Testing Modular Programs. In *ACM CoNEXT*, 2018.

[83] H. Zhu, T. Wang, Y. Hong, D. Ports, A. Sivaraman, and X. Jin. NetVRM: Virtual Register Memory for Programmable Networks. In *USENIX NSDI*, 2022.

[84] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore. NetFPGA SUME: Toward 100 Gbps as Research Commodity. *IEEE Micro*, 2014.

A   Daisy-Chain vs. Fully-AXI-L-Based Configuration

As discussed in §3.1, Menshen uses a daisy chain pipeline to configure the Menshen pipeline and uses the AXI-L [28] protocol for safety alone, i.e., to read the reconfiguration packet counter and update the bitmap during reconfiguration. Before using this daisy-chain approach, we considered a different approach based fully on the AXI-L protocol. In this approach, all configuration settings on the FPGA would be set using the AXI-L protocol via PCIe from the host instead of passing a reconfiguration packet through a daisy chain pipeline. We elected to use the daisy-chain approach instead for 2 reasons described below.

First, as one AXI-L write in Corundum can only support a 32-bit data length, we have to write $\lceil 625/32 \rceil = 20$ and $\lceil 205/32 \rceil = 7$ times for configuring one entry in the VLIW action table and CAM respectively. For our test modules, we estimate AXI-L reconfiguration time based on the write time of a single AXI-L write. As shown in Figure 12, Menshen's daisy-chain configuration is much faster than the AXI-L based method, especially for longer entries (i.e., VLIW action table). These benefits are likely to be more pronounced on a larger implementation of Menshen because the entries (both for VLIW action table and CAM) will be even longer in that case. Second, the daisy-chain approach is more similar in style to how programmable switch ASICs are configured today, hence, it is preferable for an eventual ASIC implementation of Menshen.
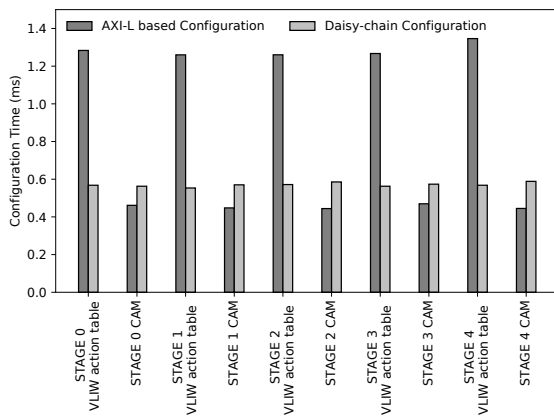
Figure 12: *Configuration time comparison for AXI-L based (estimated) and Menshen's daisy-chain configuration (measured).*

B   Isolation of ternary match tables using the Xilinx CAM IP

While our current Menshen implementation only supports exact matching, we could reuse our implementation strategy (the Xilinx CAM IP) for ternary matching as well. However, supporting isolation between the ternary match tables of multiple different modules requires some care. This is to ensure that updates to the ternary match-action rules for one module do not cause updates to the ternary match-action rules for another module.

In the case of ternary matching, the Xilinx CAM IP block uses the address of a CAM entry as the TCAM priority to determine which entry to return when there are multiple matches [31]. Concretely, the Xilinx CAM IP block can prioritize either the entry with the lowest address or the highest address. To support isolation on top of this block, first, we append the module ID (i.e., VLAN ID) to ternary match-action rules as we do currently for exact matches (§3). Second, we allocate contiguous addresses within the Xilinx CAM IP block to a particular module.

Appending the module ID ensures that a module's packets do not match any other module's match-action rules. Allocating contiguous addresses ensures that a new match-action rule can be added (or an old rule can be updated) for a module with disruption to that module's match-action rules alone—and importantly, without disturbing the rules for any other modules.[7]

C   Hardware resources in Menshen

Hardware Resource	Description
Packet Filter	A 32-bit bitmap, and a 4-byte reconfiguration packet counter
PHV	2-byte, 4-byte, 6-byte containers, each type has 8 containers a 32-byte container for platform-specific metadata
Parsing action	16 bits wide
Parser and deparser table	10 parsing actions, 160 bits wide, 32 entries deep
Key extractor table	38 bits wide, 32 entries deep
Key mask table	193 bits wide, 32 entries deep
Exact match table	205 bits wide, 16 entries deep
ALU Action	25 bits wide
VLIW action table	25 ALU actions, 625 bits wide, 16 entries deep
Segment table	16 bits wide, 32 entries deep
Stages	5
Module ID	12 bits

Table 5: *Hardware resources in Menshen*

D   Experimental setup
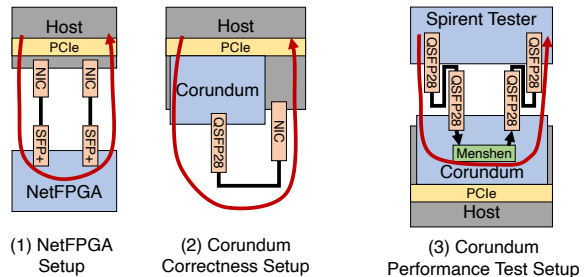
(a) *Correctness setup.*   **(b)** *Performance test setup.*
Figure 13: *Testbed setup. Red arrow shows packet flow.*

---

[7]Note that a new rule can be added to a module only if there are still empty addresses within that module's chunk of contiguously allocated addresses.

Justitia: Software Multi-Tenancy in Hardware Kernel-Bypass Networks

Yiwen Zhang*, Yue Tan*◇, Brent Stephens†, and Mosharaf Chowdhury*
*University of Michigan, ◇Princeton University, †University of Illinois at Chicago

Abstract

Kernel-bypass networking (KBN) is becoming the new norm in modern datacenters. While hardware-based KBN offloads all dataplane tasks to specialized NICs to achieve better latency and CPU efficiency than software-based KBN, it also takes away the operator's control over network sharing policies. Providing policy support in multi-tenant hardware KBN brings unique challenges – namely, preserving ultra-low latency and low CPU cost, finding a well-defined point of mediation, and rethinking traffic shapers. We present Justitia to address these challenges with three key design aspects: (i) Split Connection with message-level shaping, (ii) sender-based resource mediation together with receiver-side updates, and (iii) passive latency monitoring. Using a latency target as its knob, Justitia enables multi-tenancy policies such as predictable latencies and fair/weighted resource sharing. Our evaluation shows Justitia can effectively isolate latency-sensitive applications at the cost of slightly decreased utilization and ensure that throughput and bandwidth of the rest are not unfairly penalized.

1 Introduction

To deal with the growing demands of ultra-low latency with high throughput (message rates) and high bandwidth in large fan-out services, ranging from parallel lookups in in-memory caches [16, 30, 32] and resource disaggregation [2, 22, 52] to analytics and machine learning [1, 26, 47], kernel-bypass networking (KBN) is becoming the new norm in modern datacenters [14, 23, 43, 44, 64]. As the name suggests, with KBN, applications bypass the operating system (OS) kernel to improve performance while relieving the CPU.

There are two major trends in KBN today. *Software-based KBN* (e.g., DPDK) removes the kernel from the data path and performs packet processing in the user space. In contrast, *hardware-based KBN* (e.g., RDMA) further lowers latency by at least one order of magnitude and reduces CPU usage by offloading dataplane tasks to specialized NICs (e.g., RDMA NICs) with on-board compute.

Hardware KBN, however, takes away the operator's control over network sharing policies such as prioritization, isolation, and performance guarantees. Unlike software KBN, coexisting applications must rely on the specialized NIC to arbitrate among data transfer operations once they are posted to the hardware. We observe that existing hardware KBNs
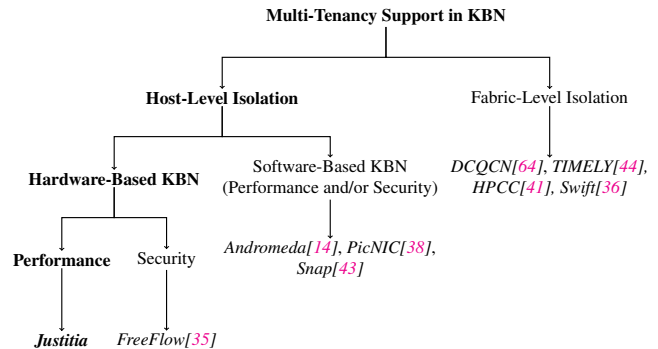
Figure 1: *Design space for multi-tenancy support in KBN.*

provide poor support for multi-tenancy. For example, even for real-world applications such as DARE [49], eRPC [32], and FaSST [31], sharing the same NIC leads to severe performance anomalies including unpredictable latency, throttled throughput (i.e., lower message rates), and unfair bandwidth sharing (§3). In this paper, we aim to address the following question: *Can we marry the benefits of software KBN with the efficiency of hardware KBN and enable fine-grained multi-tenancy support?*

Recent works have explored multi-tenancy support in large-scale software-based KBN deployments [14, 38, 43]. Their designs enforce fine-grained sharing policies such as performance and security (address space) isolation at the end hosts and pair with fabric-level solutions (e.g., congestion control) in case the network fabric becomes a bottleneck (Figure 1). However, existing software KBN solutions cannot be applied to hardware-based KBN due to three unique challenges:

1. Because host CPU is no longer involved, common CPU-based resource allocation mechanism cannot be applied. Instead, tenants issue RDMA operations with arbitrary data load at no CPU cost, which leaves no obvious point of control to exert resource mediation.

2. Hardware offloading brings packetization from user space into the NIC, disabling fine-grained user-space shaping at the packet level [27, 51].

3. It is also crucial to preserve hardware-based KBN's efficiency (i.e., single μs latency and low CPU cost[1]) while providing multi-tenancy support.

We present Justitia, a software-only solution that enables

---

[1]This does not apply to applications that aim for low latency or high message rates and busy spin their cores for maximum performance.

multi-tenancy support in hardware-based KBN, to address the aforementioned challenges (§4). Our key idea is to introduce an efficient software mediator in front of the NIC that can implement performance-related multi-tenancy policies – including (1) fair/weighted resource sharing and (2) predictable latencies while maximizing utilization or a mix of the two. Given that RDMA is the primary hardware-based KBN implementation today, in this paper, we specifically focus our solutions on RDMA NICs (RNICs).

Enabling fine-grained sharing policies in RDMA requires an efficient way of managing RNIC resources (i.e., link bandwidth and execution throughput). To this end, we propose *Split Connections* that decouple a tenant application's intent from its actuation and introduces a point of resource mediation. Justitia mediates RNIC resources by combining the benefits of sender-based and receiver-based design. RDMA operations are split and paced at the sender side before placing them onto the RNIC; receiver-side updates are collected to avoid spurious resource allocation caused by either incast or RDMA READ contention. Shaping is performed at the message level, where message sizes and their pacing rate are adjusted dynamically based on the current policy in use. By splitting RDMA connections, Justitia can effectively manage tenants' connections to consume RNIC resources based on the policy we set instead of letting tenants themselves compete by arbitrarily issuing RDMA operations.

To provide predictable latencies for latency-sensitive applications, Justitia introduces the concept of *reference flow* and monitors its latency instead of intercepting low-latency tenant applications. By comparing the latency measurements of many reference flows from the same sender machine to different receivers, Justitia can quickly detect (local and remote) RNIC resource contention. Given a tail latency target, Justitia maximizes RNIC resource utilization without violating the target. When the target is unachievable, based on the operator-defined policy, Justitia can choose to ensure that each of the competing n entities gets at least $\frac{1}{n}$th of one of the RNIC's two resources, extending the classic hose model of network sharing [17] to multi-resource RNICs.

We have implemented (§5) and evaluated (§6) Justitia on both InfiniBand and RoCEv2 networks. It provides multi-tenancy support among different types of applications without incurring high CPU usage (1 CPU core per host), introducing additional overheads, or modifying application codes. For example, using Justitia, DARE's tail latency improves by $3.4\times$ when running in parallel with Apache Crail [5, 60], a bandwidth-sensitive storage application, and Justitia preserves 81% of Crail's original performance. Justitia also complements RDMA congestion control protocols like DCQCN [64] while further mitigating receiver-side RNIC contention, and reduces tail latency even when the network is congested.
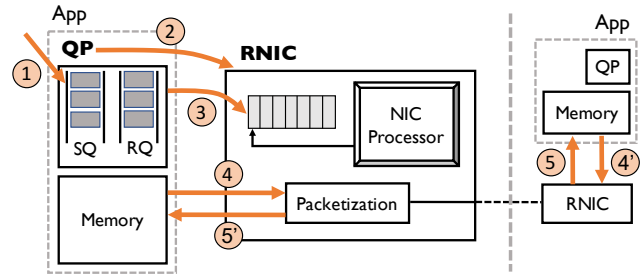
Figure 2: *Overview of host-RNIC interaction when posting (i) an RDMA WRITE operation (① → ②. → ③ → ④ → ⑤) and (ii) an RDMA READ operation (① → ② → ③ → ④' → ⑤').*

2 Background

Recent works [36, 38] have discovered unpredictable latencies due to end-host resource contention, but their primary focus is on receiver-side engine congestion in software-based KBN. In this work, we aim to emphasize that *sender-side resource contention* in hardware-based KBN such as RDMA can also lead to severe performance degradation when multiple tenants coexist. An ideal solution should address both sender- and receiver-side issues. In this section, we give an overview on how an RDMA operation is performed, followed by the root cause of RDMA's lack of multi-tenancy support.

2.1 Life Cycle of an RDMA Operation

RDMA enables direct access between user-registered memory regions without involving the OS kernel, offloading data transfer tasks to the RNIC. Applications initiate RDMA operations by posting Work Requests (WRs) via Queue Pairs (QPs) to describe the messages to transmit. Figure 2 shows how an RDMA application interacts with an RNIC to initiate an RDMA operation. To start an RDMA WRITE, ① the user application place a Work Queue Element (WQE) describing the message to the Send Queue (SQ), and ② rings a door bell to notify the RNIC by writing its QP number into the corresponding doorbell register on RNIC. At this point, the user application has completed its task and offloads the rest of the work to RNIC. After the RNIC gets notified, it ③ fetches and processes the requests from the send queue, and ④ pulls the message from the user memory, splits it into packets, and sends it to the remote RNIC. Finally, the remote RNIC ⑤ writes the received message directly into the remote memory.

In the case of an RDMA READ operation, the user application again posts the WQE and notifies the RNIC to collect it (① → ③). The local RNIC then ④' notifies the remote RNIC to pull the data from remote memory, and ⑤' places the message back to local memory after de-packetizing the received packets. Despite the opposite direction of data transfer, the remote OS remains passive just as the case with an RDMA WRITE. In both cases, the sender of the RDMA operation actively controls what goes into the RNIC while the remote side stays passively unaware.[2]

---

[2]This is true even for two-sided operations that require the receiver to post WQEs to its Receive Queue before a Send Request arrives. We still

2.2 Lack of Multi-Tenancy Support

RDMA lacks multi-tenancy support for two primary reasons: (i) tenants/applications compete for multiple RNIC resources, and (ii) RNIC processes ready-to-consume message in a greedy fashion to maximize utilization. Both are related to different symptoms of the isolation issues.

Multi-Resource Contention There exist two primary resources that need to be shared on an RNIC: *link bandwidth* and *execution throughput*. Bandwidth-sensitive applications consume RNIC's link bandwidth to issue large DMA requests. Throughput-sensitive applications, on the other hand, consume RNIC's execution throughput to issue small DMA requests in batches. Latency-sensitive applications, however, consume neither resource with the small messages they sparsely send. As we will soon show (§3), isolation anomalies can occur when applications compete for different resources.

Greedy Processing for High Utilization Although the actual RNIC implementation details are private, we can consider two hypotheses on how RNIC handles multiple requests simultaneously: either the RNIC buffers WQEs collected in ③ in Figure 2 from multiple applications and arbitrates among them using some scheduling mechanism; or it processes them in a greedy manner. When a latency-sensitive application competes with a bandwidth-sensitive application, too much arbitration in the former can cause low resource utilization (e.g., unable to catch up the line rate), whereas too little arbitration in the latter leads to head-of-line (HOL) blocking (which leads to latency variation). Our observations across all three RDMA implementations (§3), where applications using small messages are consistently affected by the ones using larger ones, suggest the latter. Note that even though receiver-side congestion can also happen during step ⑤ as pointed out in [38], both root causes can easily stem from the sender side of the operation via step ③ and thus cannot be ignored. We elaborate on how Justitia mitigates both sender- and receiver-side issues in Section 4.

3 Performance Isolation Anomalies in RDMA

This section establishes a baseline understanding of sharing characteristics in hardware KBN and identifies common isolation anomalies across different RDMA implementations with both microbenchmarks (§3.1) and highly optimized, state-of-the-art RDMA-based applications (§3.2).

To study RDMA sharing characteristics among applications with different objectives, we consider three major types of RDMA-enabled applications:

1. *Latency-Sensitive:* Sends small messages and cares about the individual message latencies.
2. *Throughput-Sensitive:* Sends small messages in batches to maximize the number of messages sent per second.

---

consider the receiver as passive because it can only control where to place a message but cannot control *when* a message will arrive.
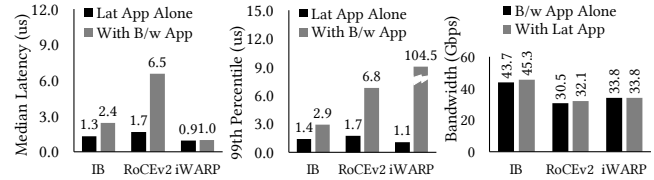
(a) *Latency App (Med)* (b) *Latency App (99th)* (c) *Bandwidth App*

Figure 3: *Latency-sensitive applications require ioslation against bandwidth-sensitive applications.*
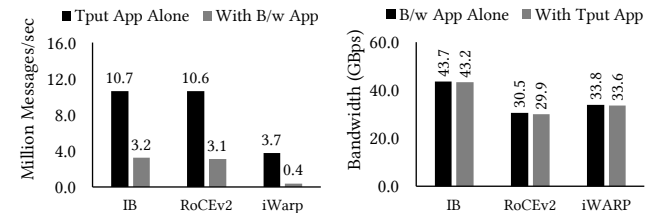
(a) *Throughput App* (b) *Bandwidth App*

Figure 4: *Throughput-sensitive application requires isolation from bandwidth-sensitive applications.*

3. *Bandwidth-Sensitive:* Sends large messages with high bandwidth requirements.

Summary of Key Findings:

- Both latency- and throughput-sensitive applications need isolation from bandwidth-sensitive applications (§3.1.1).
- If only latency- or throughput-sensitive applications (or a mix of the two types) compete, they are isolated from each other (§3.1.2).
- Multiple bandwidth-sensitive applications can lead to unfair bandwidth allocations depending on their message sizes (§3.1.3).
- Highly optimized, state-of-the-art RDMA-based systems also suffer from the anomalies we discovered (§3.2).

In the rest of this section, we describe our experimental settings and elaborate on these findings.

3.1 Observations From Microbenchmarks

We performed microbenchmarks between two machines with the same type of RNIC, where both are connected to the same RDMA-enabled switch. For most of the experiments, we used 56 Gbps Mellanox ConnectX-3 Pro for InfiniBand, 40 Gbps Mellanox ConnectX-4 for RoCEv2, and 40 Gbps Chelsio T62100 for iWARP; 10 and 100 Gbps settings are described similar. More details on our hardware setups are in Table 1 of Appendix A.

Our benchmarking applications are written based on Mellanox perftest [61] and each of them uses a single Queue Pair. Unless otherwise specified, latency-sensitive applications in our microbenchmarks send a continuous stream of 16B messages, throughput-sensitive ones send a continuous stream of batches with each batch having 64 16B messages, and bandwidth-sensitive applications send a continuous stream of 1MB messages. Although all applications send messages

using RDMA WRITEs over reliable connection (RC) QPs in the observations below, other verbs show similar anomalies as well. We defer the usage and discussion of hardware virtual lanes to Section 6.3.

3.1.1 Both Latency- and Throughput-Sensitive Applications Require Isolation

The performance of the latency-sensitive applications deteriorate for all RDMA implementations (Figure 3). Out of the three implementations we benchmarked, InfiniBand and RoCEv2 observes $1.85\times$ and $3.82\times$ degradations in median latency and $2.23\times$ and $4\times$ at the 99th percentile. While iWARP performs well in terms of median latency, its tail latency degrades dramatically ($95\times$).

Throughput-sensitive applications also suffer. When a background bandwidth-sensitive application is running, the throughput-sensitive ones observe a throughput drop of $2.85\times$ or more across all RDMA implementations (Figure 4). Note that in our microbenchmark with 1 QP per application, throughput-sensitive applications that consume NIC execution throughput hit the bottleneck. This does not imply RNIC always favors link bandwidth over execution throughput. We notice RNIC bandwidth starts to become the bottleneck when there exists $4\times$ more throughput-sensitive applications.

More importantly, both latency- and throughput-sensitive applications experience more severe performance degradations (e.g., 139X worse latency with the presence of 16 bandwidth applications) as more bandwidth-sensitive applications join the competition, which is prevalent in shared datacenters [23, 64]. Appendix B.1 provides more details.

3.1.2 Latency-Sensitive Applications Coexist Well; So Do Throughput-Sensitive Ones

We observe no obvious anomalies among latency- or throughput-sensitive applications, or a mix of the two types. Detailed results can be found in Appendix B.

3.1.3 Bandwidth-Sensitive Applications Hurt Each Other

Unlike latency- and throughput-sensitive applications, bandwidth-sensitive applications with different message sizes do affect each other.Figure 5 shows that a bandwidth-sensitive application using 1MB messages receive smaller share than one using 1GB messages. The latter receives $1.42\times$, $1.22\times$ and $1.51\times$ more bandwidth in InfiniBand, RoCEv2, and iWARP, respectively.

3.1.4 Anomalies are Present in Faster Networks Too

We performed the same benchmarks on 100 Gbps InfiniBand, only to observe that most of the aforementioned anomalies are still present. Appendix C.1 has the details.

3.2 Isolation Among Real-World Applications

In this section, we demonstrate how real RDMA-based systems fail to preserve their performance in the presence of the
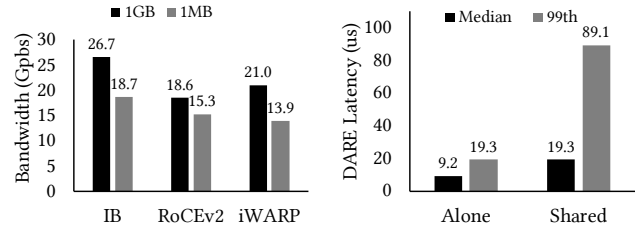
Figure 5: *Anomalies among Bandwidth-sensitive applications with different message sizes.*

Figure 6: *Latency of DARE's Put and Get operations when coexisting with Apache Crail's storage traffic.*

aforementioned anomalies.

Specifically, we performed experiments with Apache Crail [5, 60] and DARE [49]. Crail is a bandwidth-hungry distributed data storage system that utilizes RDMA. In contrast, DARE is a latency-sensitive system that provides high-performance replicated state machines through the use of a strongly consistent RDMA-based key-value store.

In these experiments, we deployed DARE in a cluster of 4 nodes with 56 Gbps Mellanox ConnectX-3 Pro NIC on InfiniBand with 64GB memory. Crail is deployed in the same cluster with one node running the namenode and one other node running the datanode.

To evaluate the performance of Crail, we launch 8 parallel writes (each to a different file) in Crail's data storage with the chunk size of the data transfer configured to be 1MB, and we measure the application-level throughput reported by Crail. To evaluate the performance of DARE, one DARE client running on the same server as the namenode of Crail issues PUT and GET operations (each PUT is followed by a GET) to the DARE server on the other 3 nodes with a sweep of message sizes from 8 byte to 1024 bytes, and we measure the application-level latency reported by DARE.

Figure 6 plots the latency of DARE's queries with and without the presence of Crail. In this experiment, we observe a $4.6\times$ increase in DARE's tail latency. Additionally, regardless of whether it is competing with DARE, Crail's total write throughput stays at 51.1 Gbps.

Besides DARE, highly-optimized RDMA-based RPC system such as FaSST [31] and eRPC [32] also suffer from isolation anomalies caused by unmanaged resource contention on RNICs. In fact, when background bandwidth-heavy traffic is present, FaSST's throughput experiences a 74% drop (Figure 32) and eRPC's tail latency increases by $40\times$ (Figure 33). More details can be found in Appendix C.2.

3.3 Congestion Control is not Sufficient

To demonstrate that DCQCN [64] and PFC are not sufficient to solve these anomalies, we performed the benchmarks again with PFC enabled at both the NICs and switch ports, DCQCN [64] enabled at the NICs, and ECN markings enabled on a Dell 10 Gbps Ethernet switch (S4048-ON). In these experiments, latency- and throughput-sensitive applications still suffer unpredictably (Section 6.3 has detailed results). This is because DCQCN focuses on fabric-level isolation whereas

the observed anomalies happen at the end host due to RNIC resource contention (§2.2).

4 Justitia

Justitia enables multi-tenancy in hardware-based KBN, with a specific focus on enabling two performance-related policies: (1) fair/weighted resource sharing, or (2) predictable latencies while maximizing utilization, or a mix of the two. Note that we restrict our focus on a *cooperative* datacenter environment in this paper and defer strategyproofness [20, 21, 50] to mitigate adversarial/malicious behavior to future work.

Granularity of Control: We define a flow to be a stream of RDMA messages between two RDMA QPs. Justitia can be configured to work either at the flow granularity or at the application granularity by considering all flows between two applications as a whole.[3] In this paper, by default, we set Justitia's granularity of control to be at the application level to focus on application-level performance.

4.1 Key Design Ideas

Justitia resolves the unique challenges of enabling multi-tenancy in hardware KBN with five key design ideas.

- *Tenant-/application-level connection management:* To prevent tenants from hogging RNIC resources by issuing arbitrarily large messages or creating a large number of active QPs at no cost, Justitia provides a tenant-level connection management scheme by adding a shim layer between tenant applications and the RNIC. Tenant operations are handled by Justitia before arriving at the RNIC.

- *Sender-based proactive resource mediation:* Justitia proactively controls RNIC resource utilization at the sender side. This is based on the observation that the sender of an RDMA operation – that decides when an operation gets initiated, how large the message is, and in which direction the message flows – has active control over every aspect of the transmission while the other side of the connection remains passive. Such sender-based control can react before the RNIC takes over and maintain isolation by directly controlling RNIC resources.

- *Dynamic receiver-side updates:* Pure sender-based approaches can sometimes lead to spurious resource allocation when multiple senders coexist but are unaware of each other. Justitia leverage receiver-side updates to provide information (e.g., the arrival or departure of an application) back to the senders to react correctly when a change in the setting happens.

- *Passive latency monitoring:* Instead of actively measuring each application's latency, which can introduce high overhead, Justitia uses passive latency monitoring by issuing *reference flows* to detect RNIC resource contention.

---

[3]Each granularity has its pros and cons when it comes to performance isolation, without any conclusive answer on the right one [46].
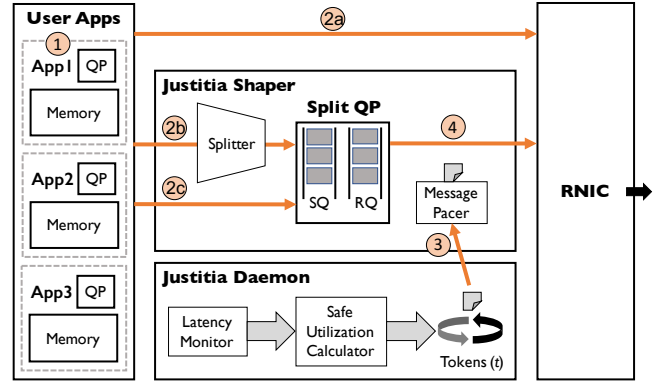
Figure 7: *Justitia architecture. Bandwidth- and throughput-sensitive applications are shaped by tokens generated at a regular interval by Justitia. Latency-sensitive ones are not paced at all.*

- *Message-level shaping with splitting:* Justitia performs shaping at the message level to suit RDMA's message-oriented transport layer. At the message level, it is easy to apply specific strategies to control how messages enter the RNIC based on their sizes and the resource they consume. Large messages are split into roughly equal-sized sub-messages or chunks to (i) avoid a single message requesting too many RNIC resources; (ii) facilitate network sharing policies such as fair/weighted bandwidth share; and (iii) mitigate HOL Blocking for latency-sensitive applications.

4.2 System Overview

Figure 7 presents a high-level system overview of Justitia handling an RDMA WRITE operation (to compare with Figure 2). Each machine has a Justitia daemon that performs latency monitoring and proactive rate management, and applications create QPs using the existing API to perform RDMA communication. Justitia relies on applications to optionally identify their application type. By default, they are treated as bandwidth-sensitive. VMs, containers, bare-metal applications, and SR-IOV are all compatible with the design of Justitia.

As before, the user application starts an RDMA WRITE operation by ① posting a WQE into the Send Queue. Latency-sensitive applications will ②a bypass Justitia and directly interact with the RNIC as shown in Figure 2. The other two types of applications will enter Justitia's shaper. The Splitter will ②b split the big message from a bandwidth-sensitive applications equally into sub-messages or ②c do nothing given a small message from a throughput-sensitive application. We introduce *Split Connection* – and corresponding split queue pair (Split QP) – to handle the messages passed through the Splitter. Before sending out the message, it ③ asks the daemon to fetch a token from Justitia, which is generated at a rate to maximize RNIC resource utilization consumed by resource-hungry applications. Once the token is fetched, the Split QP ④ posts a WQE for the sub-message into its SQ and rings the door bell to notify the RNIC. The RNIC then grabs

the WQE from Split QP, issue a DMA read for the actual data in application's memory region, and sends the message to the remote side (arrows not shown in the figure). Steps ③ and ④ repeat until all messages in the Split QP have been processed. The implementation details of Split QP is in Section 5.1.

The Justitia daemon in Figure 7 is a background process that performs latency monitoring and proactive rate management to maximize RNIC resource utilization when latency target is met.

4.3 Justitia Daemon

Justitia daemon performs two major tasks: (i) proactively manages rate of all bandwidth- and throughput-sensitive applications using the hose model [17]; (ii) ensures predictable performance for latency-sensitive applications while maximizing RNIC resource usage.

4.3.1 Minimum Guaranteed Rate

Justitia enforces rate based on the classic hose model [17], and always maintains a minimum guaranteed rate R_{min}:

$$R_{min} = \frac{\sum w_B^i + \sum w_T^i}{\sum w_B^i + \sum w_T^i + \sum w_L^i} \times MaxRate$$

where w_X^i represents the weight of application i of type X (i.e., bandwidth-, throughput-, or latency-sensitive), and *MaxRate* represents the maximum RNIC bandwidth or maximum RNIC throughput (both are pre-determined on a per-RNIC basis) depending on the type of the application. The idea of R_{min} is to recognize the existence of latency-sensitive applications, and provide isolation for them by *taking out their share from the RNIC resources which otherwise they cannot acquire by themselves*. In the absence of latency-sensitive applications (i.e., $\sum w_L^i = 0$), R_{min} is equivalent to *MaxRate*, and all the resource-hungry applications share the entire RNIC resources. If all applications have equal weights, and there exist B bandwidth-, T throughput-, and L latency-sensitive applications, R_{min} can be simplified as $\frac{B+T}{B+T+L} \times MaxRate$.

In the presence of a large number of latency-sensitive applications, R_{min} could be really small, essentially removing RNIC resource guarantee. To accommodate such cases, one can fix $L = 1$ no matter how many latency-sensitive applications join the system since they do not consume much of RNIC's resources. We find this setting works well in practice (§6.4) and make it the default option for Justitia.

With R_{min} provided, Justitia then *maximizes RNIC's safe resource utilization* (which we denote *SafeUtil*) until the performance of latency-sensitive applications crosses the target tail latency (*Target₉₉*).

4.3.2 Latency Monitoring via Reference Flows

Justitia does not interrupt or interact with latency-sensitive applications because (i) they cannot saturate either of the two RNIC resources, and (ii) interrupting them fails to preserve RDMA's ultra-low latency.

Pseudocode 1 Maximize *SafeUtil*

```
 1: procedure ONLATENCYFLOWUPDATE(L, Estimated₉₉)
 2:   if L = 0 then        ▷ Reset if no latency-sensitive applications
 3:     SafeUtil = MaxRate
 4:   else
 5:     if Estimated₉₉ > Target₉₉ then
 6:       SafeUtil = max(SafeUtil/2, Rmin)
 7:     else
 8:       SafeUtil = SafeUtil + 1
 9:     end if
10:   end if
11:   τ = Token_Bytes/ SafeUtil
12: end procedure
```

Instead, whenever there exists one or more latency-sensitive applications to particular receiving machine, Justitia maintains a *reference flow* to that machine which keeps sending 10B messages to the same receiver as the latency-sensitive applications in periodic intervals (by default, *RefPeriod* = 20 μs) to estimate the 99th percentile (*Estimated₉₉*) latency for small messages. By monitoring its own reference flow, Justitia does not need to wait on latency-sensitive applications to send a large enough number of sample messages for accurate tail latency estimation. It does not add additional delay by directly probing those applications either.

Given the stream of measurements, Justitia maintains a sliding window of the most recent *RefCount* (=10000) measurements for a reference flow estimate its tail latency.

4.3.3 Maximizing *SafeUtil*

Using the selected latency measurement from the reference flow(s), Justitia maximizes *SafeUtil* based on the algorithm shown in Pseudocode 1. To continuously update *SafeUtil*, Justitia uses a simple AIMD scheme that reacts to *Estimated₉₉* every *RefPeriod* interval as follows. If the estimation is above *Target₉₉*, Justitia decreases *SafeUtil* by half; *SafeUtil* is guaranteed to be at least R_{min}. If the estimation is below *Target₉₉*, Justitia slowly increases *SafeUtil*. Because *SafeUtil* ranges between R_{min} to the total RNIC resources and latency-sensitive applications are highly sensitive to too high a utilization level, our conservative AIMD scheme, which drops utilization quickly to meet *Target₉₉*, works well in practice.

To determine the value of *Target₉₉*, we constructs a latency oracle that performs pair-wise latency measurement by issuing reference flows across all the nodes in the cluster when there is no other background. Microsoft applies a similar approach in [24], which is shown to work well in estimating steady-state latency in the cluster. We adopt this approach to give a good estimate of the latency target under well-isolated scenarios.

4.3.4 Token Generation And Distribution

Justitia uses multi-resource tokens to enforce *SafeUtil* among the B bandwidth- and T throughput-sensitive applications in a fair or weighted-fair manner. Each token represents a fixed

---

amount of bytes ($Token_{Bytes}$) and a fixed number of messages ($Token_{Ops}$). In other words, the size of $Token_{Bytes}$ determines the chunk size a message from bandwidth-sensitive application is split into. A token is generated every τ interval, where the value of τ depends on *SafeUtil* as well as on the size of each token. For example, given 48 Gbps application-level bandwidth and 30 Million operations/sec on a 56 Gbps RNIC, if $Token_{Bytes}$ is set to 1MB, then we set $Token_{Ops}$ =5000 ops and τ =167 μs.

Justitia daemon continuously generates one token every τ interval and distributes it among the active resource-hungry applications in a round-robin fashion based on application weights w_X^i. When $w_X^i = 1$ for all applications, Justitia enforces traditional max-min fairness; otherwise, it enforces weighted fairness. Each application independently enforces its rate using one of the shapers described below.

4.4 Justitia Shapers

Justitia shapers – implemented in the RDMA driver – enforce utilization limits provided by the Justitia daemon-calculated tokens. There are two shapers in Justitia: one for bandwidth- and another for throughput-sensitive applications.

Split Connection Justitia introduces the concept of a Split Connection to provide an interface to coordinate between tenant applications and the RNIC. It consists of a message splitter and custom *Split QPs* (§5.1) to initiate RDMA operations for tenants. Each application's Split Connection cooperate with Justitia daemon to pace split messages transparently.

Shaping Bandwidth-Sensitive Applications. This involves two steps: *splitting* and *pacing*. For any bandwidth-sensitive application, Justitia *transparently* divides any message larger than $Token_{Bytes}$ into $Token_{Bytes}$-sized chunks to ensure that the RNIC only sees roughly equal-sized messages. Splitting messages for diverse RDMA verbs – e.g., one-sided vs. two-sided – requires careful designing (§5.1).

Given chunk(s) to send, the pacer requests for token(s) from the Justitia daemon by marking itself as an active application. Upon receiving a token, it transfers chunk(s) until that token is exhausted and repeats until there is nothing left to send. The application is notified of the completion of a message only after all of its split messages have been transferred.

Batch Pacing for Throughput-Sensitive Applications. These applications typically deal with (batches of) small messages. Although there is no need for message splitting, pacing individual small messages requires the daemon to generate and distribute a large number of tokens, which can be CPU-intensive. Moreover, for messages as small as 16B, such fine-grained pacing cannot preserve RDMA's high message rates.

To address this, Justitia performs *batch pacing* enabled by Justitia's multi-resource token. Each token grants an application a fixed batch size ($Token_{Ops}$) that it can send together before receiving the next token. Batch pacing on throughput-sensitive applications removes the bottleneck on token gener-
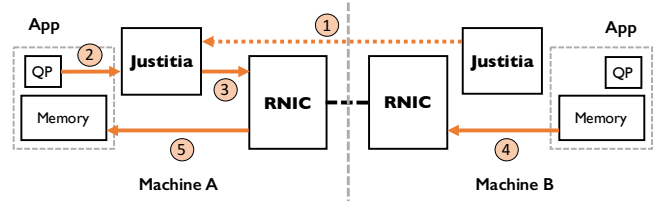
Figure 8: *How Justitia handles READs via remote control.*

ation and distribution; it also relieves daemon CPU cost with a unified token bucket.

Mitigating Head-of-Line Blocking. One of the foremost goals of Justitia is to mitigate HOL blocking caused by the bandwidth-sensitive applications to provide predictable latencies. To achieve this goal, we need to split messages into smaller chunks and pace them at a certain rate (enforcing *SafeUtil*) with enough spacing between them to minimize the blocking. However, this simple approach creates a dilemma. On the one hand, too large a chunk may not resolve HOL Blocking. On the other hand, too small a chunk may not be able to reach *SafeUtil*. It also leads to increased CPU overhead from using a spin loop to fetch tokens generated in a very short period in which context switches are not affordable. This is a manifestation of the classic performance isolation-utilization tradeoff. We discuss how to pick the chunk size in Section 5.2.

4.5 Dynamic Receiver-Side Updates

Justitia relies on receiver-side updates to coordinate among multiple senders to avoid spurious allocation of RNIC resources. The benefits of this design is three-fold: (i) it coordinates with multiple senders to provide the correct resource allocation; (ii) it keeps track of RDMA READ issued which can collide with applications issuing RDMA WRITE in the opposite direction; (iii) it mitigates receiver-side engine congestion by rate-limiting senders with the correct fan-in information.

The updates are communicated among Justitia Daemons only when a change in the application state happened to a certain receiver (i.e., an arrival or an exit of an application) is detected. Two-sided operations, SEND and RECV, are selected in such case so that the daemon gets notified when an update arrives. Once a change is detected by a sender, it informs the receiver, which then broadcasts the change back to all the senders it connects to so that they can update the correct R_{min}. In such case, R_{min} considers remote resource-hungry application count as part of the total share. If the local daemon has not issued a reference flow and a remote latency-sensitive applications launches to the receiver, the daemon will start a new reference flow to start latency monitoring.

Handling READs RDMA specification allows remote machines to read from a local machine using the RDMA READ verb. RDMA READ operations issued by machine *A* to read data from machine *B* compete with all sending operations (e.g., RDMA WRITE) from machine *B*. Consequently, Justitia must handles remote READs as well.
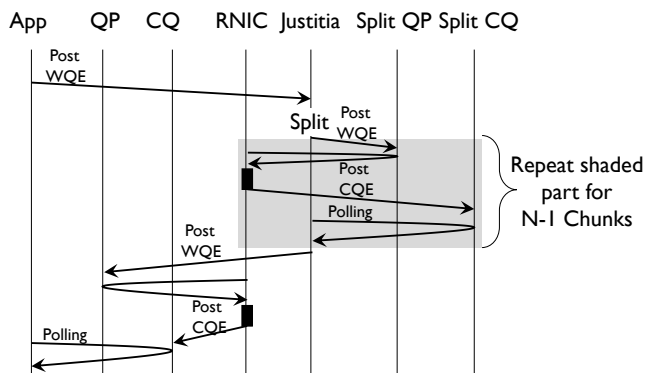
Figure 9: *High-level overview of transparent message splitting in Justitia for one-sided verbs using Split QP. Times are not drawn to scale. Two-sided verbs involve extra bookkeeping.*
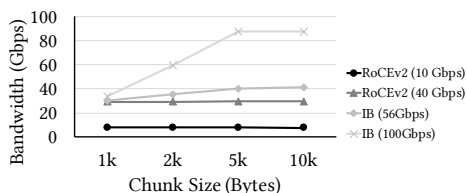

Figure 10: *Maximum achievable bandwidth vs. chunk sizes.*

In such a case, the receiver of the READ operation, machine *B*, sends the updated guaranteed utilization R_{min}, with the updated count of senders including remote READ applications) as shown in ① in Figure 8. After *A* receives that utilization, it operates RDMA READ by interact with Justitia normally via ② → ⑤ and enforces the updated rate.

5 Implementation

We have implemented the Justitia daemon as a user-space process in 3,100 lines of C, and the shapers are implemented inside individual RDMA drivers with 5,200 lines of C code. Our current implementation focuses on container/bare-metal applications. Justitia code is available at https://github.com/SymbioticLab/Justitia.

5.1 Transparently Splitting RDMA Messages

Justitia splitter transparently divides large messages of bandwidth-sensitive applications into smaller chunks for pacing. Our splitter uses a custom QP called a *Split QP* to handle message splitting, which is created when the original QP of a bandwidth-sensitive flow is created. A corresponding *Split CQ* is used to handle completion notifications. A custom completion channel is used to poll those notifications in an event-triggered fashion to preserve low CPU overhead.

To handle one-sided RDMA operations, when detecting a message larger than $Token_{Bytes}$, we divide the original message into chunks and only post the last chunk to the application's QP (Figure 9). The rest of the chunks are posted to the Split QP. Split QP ensures all chunks have been successfully transferred before the last chunk handled by the application's QP. The two-sided RDMA operations such as SEND are handled in a similar way, with additional flow control messages for the chunk size change and receive requests

(a) *Latency-sensitive App*   (b) *Bandwidth App*
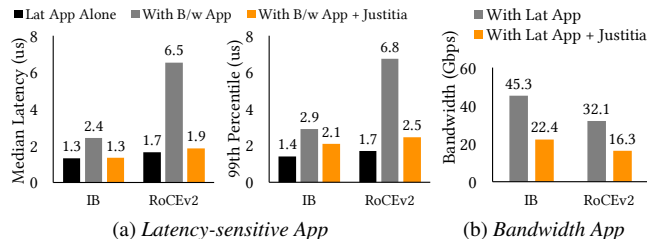
Figure 11: *Performance isolation of a latency-sensitive application running against a bandwidth-sensitive one.*

(a) *Latency-sensitive App*   (b) *Bandwidth App*
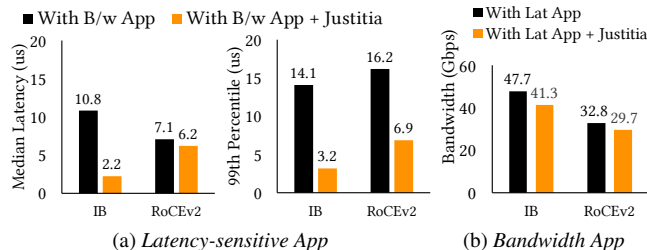
Figure 12: *Latency of a latency-sensitive application running against a bandwidth-sensitive application with 4 QPs with a relaxed latency target (10 μs).*

to be pre-posted at the receiver side.

5.2 Determining Token Size for Bandwidth Target

One of the key steps in determining *SafeUtil* is deciding the size of each token. Because the RNIC can become throughput-bound for smaller messages instead of bandwidth-bound, we cannot use arbitrarily small messages to resolve HOL blocking. At the same time, given a utilization target, we want to use the smallest $Token_{Bytes}$ value to achieve that target to reduce HOL blocking while maximizing utilization.

Instead of dynamically determining it using another AIMD-like process, we observe that (i) this is an RNIC-specific characteristic and (ii) the number of RNIC types is small. With that in mind, we maintain a pre-populated dictionary to store the smallest token size that can saturate a given rate (to enforce *SafeUtil*) when sending in a paced batch for different latency targets; Justitia simply uses the mappings during runtime. When latency-sensitive applications are not present, a large token size (1MB) is used. Otherwise, Justitia looks up the token size in the dictionary based on the current *SafeUtil* value. This works well since the lower the *SafeUtil* is, the smaller the chunk size it requires to achieve such *SafeUtil*, and the better it helps mitigating HOL blocking. Based on our microbenchmarks (Figure 10), we pick 5KB as the chunk size when latency-sensitive applications are present.

6 Evaluation

In this section, we evaluate Justitia's effectiveness in providing multi-tenancy support among latency-, throughput-, and bandwidth-sensitive applications on InfiniBand and RoCEv2. To measure latency, we perform 5 consecutive runs and present their median. We do not show error bars when they are too close to the median.
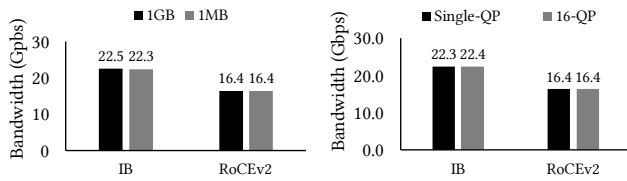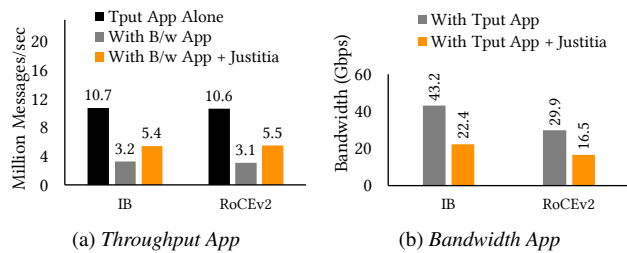
(a) *Different message sizes*   (b) *Single-QP vs. 16-QP*

Figure 13: *Fair bandwidth share of bandwidth-sensitive applications. (a) different message sizes. (b) different number of QPs.*
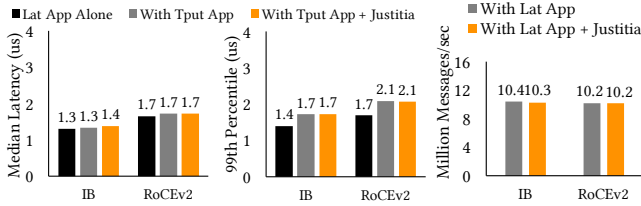
(a) *Latency-sensitive App*   (b) *Throughput App*

Figure 14: *Performance isolation of a latency-sensitive application running against a throughput-sensitive application.*

(a) *Throughput App*   (b) *Bandwidth App*

Figure 15: *Performance isolation of a throughput-sensitive application running against a bandwidth-sensitive application.*

Our key findings can be summarized as follows:

- Justitia can effectively provide multi-tenancy support highlighted in Section 3 both in microbenchmarks and at the application-level (§6.1).

- Justitia scales well to a large number of applications and works for a variety of settings (§6.2); it complements DCQCN and hardware virtual lanes (§6.3).

- Justitia's benefits hold with many latency- and bandwidth-sensitive applications (§6.4), in incast scenarios (§6.5), and under unexpected network congestion (§6.6).

A detailed sensitivity analysis of Justitia parameters can be found in Appendix D.

6.1 Providing Multi-Tenancy Support

We start by revisiting the scenarios from Section 3 to evaluate how Justitia enables sharing policies among different RDMA applications. We use the same setups as those in Section 3. Unless otherwise specified, we set $Target_{99}$ =2 μs on both InfiniBand and RoCEv2 for the latency-sensitive applications. Justitia works well in 100 Gbps networks too (Appendix C.1). Unless otherwise specified, R_{min} with all applications sharing the same weights is enforced as a default policy.
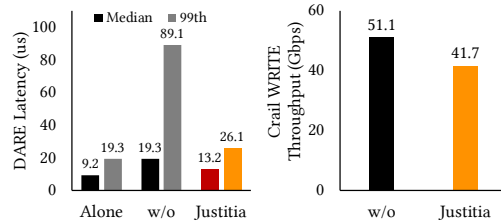
(a) *DARE Latency*   (b) *CRAIL Throughput*

Figure 16: *[InfiniBand] Performance isolation of DARE running against Crail.*
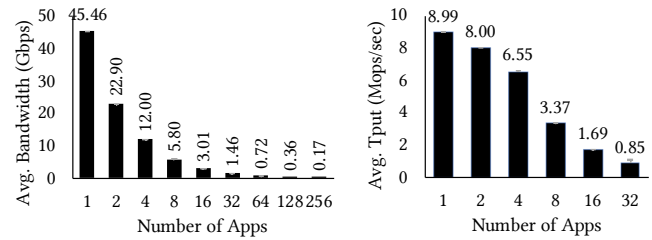
(a) *Bandwidth-sensitive*   (b) *Throughput-sensitive*

Figure 17: *[InfiniBand] Justitia scales to a large number of applications and still provides equal share. The error bars represent the minimum and the maximum values across all the applications.*
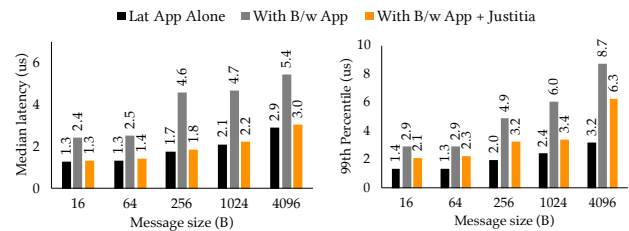
Figure 18: *[InfiniBand] Latency-sensitive applications with different message sizes competing against a bandwidth-sensitive app.*

Predictable Latency   Latency-sensitive applications are affected the most when they compete with a bandwidth-sensitive application. In the presence of Justitia, both median and tail latencies improve significantly in both InfiniBand and RoCEv2 (Figure 11a). Due to the enforcement of R_{min}, the bandwidth-sensitive application is receiving half of the capacity (Figure 11b).

Next we evaluate how Justitia performs when the latency target is set to a relaxed value ($Target_{99}$ =10 μs) that can be easily met (Figure 12). For a slightly high $Target_{99}$, Justitia maximizes utilization, illustrating that splitting and pacing are indeed beneficial.

Fair Bandwidth and Throughput Sharing   Justitia ensures that bandwidth-sensitive applications receive equal shares regardless of their message sizes and number of QPs in use (Figure 13) with small bandwidth overhead (less than 6% on InfiniBand and 2% on RoCEv2). The overhead becomes negligible when applying Justitia to throughput- or latency-sensitive applications (Figure 14).

Justitia's benefits extends to the bandwidth- vs through-sensitive application scenario as well. In this case, it ensures

(a) *Latency-sensitive App*   (b) *Bandwidth App*
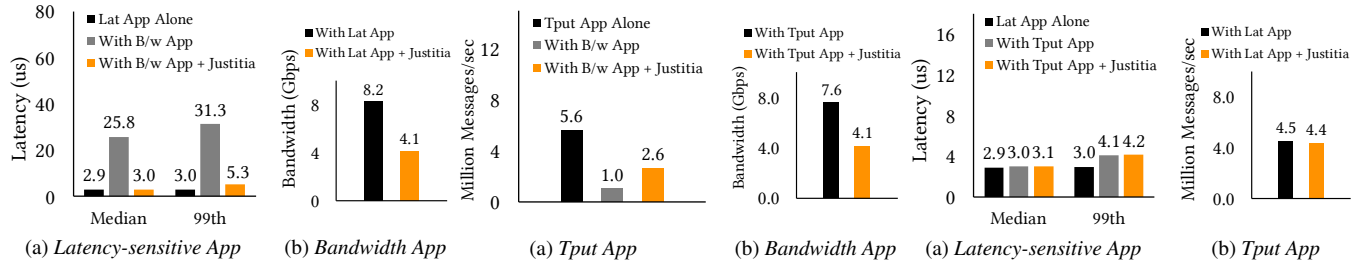
Figure 19: *[DCQCN] Latency-sensitive application against a bandwidth-sensitive one.*

(a) *Tput App*   (b) *Bandwidth App*

Figure 20: *[DCQCN] Throughput-sensitive app against a bandwidth-sensitive one.*

(a) *Latency-sensitive App*   (b) *Tput App*

Figure 21: *[DCQCN] Latency-sensitive application against a throughput-sensitive one.*

that both receive roughly half of their resources. Figure 15 illustrates this behavior. In both InfiniBand and RoCEv2, the throughput-sensitive application is able to achieve half of its original message rate of itself running alone (Figure 15a). The bandwidth-sensitive application, on the other hand, is limited to half its original bandwidth as expected (Figure 15b).

Justitia and Real-World RDMA Applications   To demonstrate that Justitia can isolate highly optimized real-world applications, we performed experiments with DARE and Crail. Thanks to Justitia's high transparency, we did not need to make any source code changes in Crail (given it is bandwidth-sensitive by default), and we only changed DARE by marking it as latency-sensitive.

From these experiments, we find that Justitia improves isolation for latency-sensitive applications while also preserving high bandwidth of the background storage application. Figure 16 plots the performance of DARE and Crail after applying Justitia with the same setting as in Section 3.2. We observe that, with Justitia, DARE achieves performance that is close to running in isolation even when running alongside Crail, and Justitia improves DARE's tail latency performance by 3.4× when compared to the baseline scenario while Crail also achieves 81% of its original throughput performance. This is close to the expected throughput of $\frac{8}{9}$ of Crail's original throughput since in this experiment Justitia treats the 8 parallel writes on top of Crail as separate applications.

Justitia improves performance isolation of FaSST by 2.5× in throughput and eRPC by 32.2× in tail latency. More details can be found in Appendix C.2.

6.2   Justitia Deep Dive

Scalability and Rate Conformance   Figure 17a shows that as the number of bandwidth-sensitive applications increases, all applications receive the same amount of bandwidth using Justitia with total bandwidth close to the line rate. Justitia also ensures that all throughput-sensitive application send roughly equal number of messages (Figure 17b).

CPU and Memory Consumption   Justitia daemon uses one dedicated CPU core per node to generate and distribute tokens. Its memory footprint is not significant.
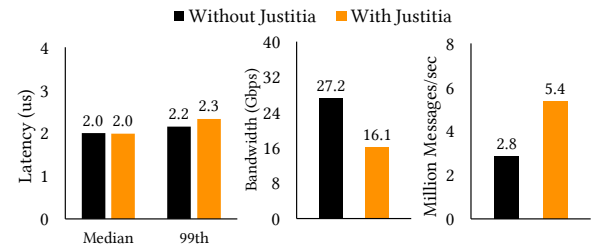
Figure 22: *[RoCEv2] A bandwidth-, throughput-, and latency-sensitive application running on two hardware priority queues at the NIC. The latency-sensitive application uses one queue, while the other two share the other queue.*

Varying Message Sizes   Justitia can provide isolation at a wide range of message sizes for latency-sensitive applications (Figure 18). The bandwidth-sensitive application receives half the bandwidth in all cases.

6.3   Justitia + X

Justitia + DCQCN   The anomalies we discover in this paper does not stem from the network congestion, but rather happens at the end hosts. We found that DCQCN falls short for latency- and throughput-sensitive applications (Figures 19, 20, 21). Justitia can complement DCQCN and improve latencies by up to 8.6× and throughput by 2.6×.

Justitia + Hardware Virtual Lanes   Although RDMA standards support up to 15 virtual lanes [7] for separating traffic classes, they only map to very few hardware shapers and/or priority queues (2 queues in our RoCE NIC) that are rarely sufficient in shared environments [3, 37]. Besides, the hardware rate limiters in the RNIC are slow when setting new rates (2 milliseconds in our setup), making it hard to use with real dynamic arrangement. Moreover, it is desirable to achieve isolation *within each priority queue*, as those hardware resources are often used to provide different levels of quality of service, within which many applications reside.

In this experiment, we show how limited number of hardware queues are insufficient to provide isolation and how Justitia can help in this scenario. we run three applications, one each for each of the three types (Figure 22). Although the latency-sensitive application remains isolated in its own class, the bandwidth- and throughput-sensitive applications compete in the same class. As a result, the latter observes throughput loss (similar to Figure 15). Justitia can effectively provide performance isolation between bandwidth- and throughput-
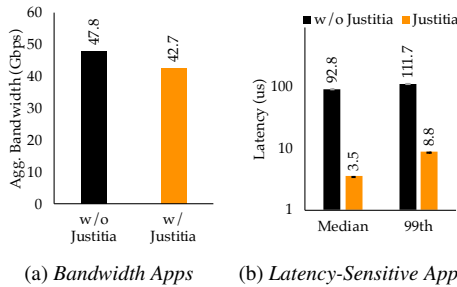
(a) Bandwidth Apps (b) Latency-Sensitive Apps

Figure 23: *[InfiniBand] Justitia isolating 8 latency-sensitive applications from 8 bandwidth-sensitive ones. Note that 8/9th of the bandwidth share is guaranteed since Justitia counts all latency-sensitive apps as one by default (§4.3.3).*
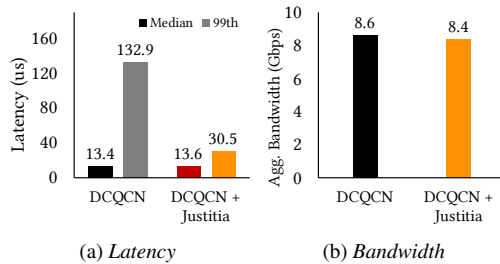

(a) Latency (b) Bandwidth

Figure 24: *[DCQCN] Incast experiment with 33 senders and a single receiver. 32 senders launch bandwidth-sensitive applications, the other sender launches a latency-sensitive application.*

sensitive applications in the shared queue.

6.4 Isolating among More Competitors

We focus on Justitia's effectiveness in isolating many applications with different requirements and performance characteristics. Specifically, we consider 8 bandwidth-sensitive applications – 2 each with message sizes: 1MB, 10MB, 100MB, and 1GB, and 8 latency-sensitive applications. We measure the latency and bandwidth when all the applications are active in Figure 23. $Target_{99}$ is set to $2\mu s$ and 20 million samples are collected for latency measurements.

Without Justitia, latency-sensitive applications suffer large performance hits: individually each application had median and 99th percentile latencies of 1.3 and 1.4 μs (Figures 3a and 3b). With bandwidth-sensitive applications, they worsen by $71.4\times$ and $79.8\times$. Justitia improves median and tail latencies of latency-sensitive applications by $26.5\times$ and $12.7\times$ while guaranteeing R_{min} among all the applications.

6.5 Handling Incast with Receiver-Side Updates

So far, we have focused on host-side RNIC contentions where the network fabric is not a bottleneck. We now evaluate how Justitia leverages receiver-side updates to handle receiver-side incast in both RoCEv2 with DCQCN and InfiniBand with its native credit-based flow control. In this experiment, 33 senders are used with the first 32 continuously launch a bandwidth-sensitive application sending 1MB messages to a single receiver. Simultaneously, the last sender launches a latency-sensitive application with messages sent to the same

(a) Topology (b) Latency (c) Bandwidth
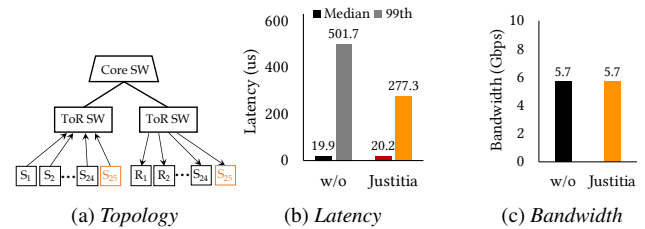
Figure 25: *[DCQCN] Justitia's performance when Inter-ToR links are congested. Justitia achieves the same bandwidth performance because the total amount of bandwidth share on S_{25} is smaller than SafeUtil due to other traffic flowing in the fabric.*

receiver. As described in Section 4.5, Justitia daemon at the receiver sends updates to all the senders whenever a sender application starts or exits, resulting in $\frac{1}{32}$-th of line rate guaranteed at each of the first 32 senders.

Figure 24 plots the results of this experiment, which show that Justitia still reduces tail latency even after the impact of fabric-level congestion on the reference flow latency measurements. Since the monitored latency misses the target, all the bandwidth-sensitive applications send at the minimum guaranteed rate. However, Justitia still achieves high aggregate bandwidth because this is greater than the fair share. This shows that Justitia complements congestion control and further improves the performance of latency-sensitive applications by mitigating receiver-side RNIC congestion.

We have also included a discussion on frequently asked questions regarding reference flows' impact in large-scale incast scenarios in Appendix E.4.

6.6 Justitia with Unexpected Network Congestion

When there is congestion inside the network, all traffic flowing through the network will experience increased latency, including the packets generated by Justitia as latency signals. Because today's switches and NICs do not report their individual contributions to end-to-end latency, Justitia cannot tell them apart. However, in practice, this is not a problem because the same response is appropriate in both scenarios.

To evaluate how Justitia performs under such cases, we performed experiments utilizing two interconnected ToR switches on CloudLab [11]. There are servers attached to each ToR switch, and every server has a line rate of 10Gbps. The experiment topology is shown in Figure 25a. In this topology, there is a third core switch that connects to each of the ToR switches with a link with a capacity of 160 Gbps. In this experiment, we enable DCQCN at all the servers and ECN marking at the ToR switches in the cluster.

To create a congested ToR uplink, we launch 24 bandwidth-sensitive applications each issuing 1MB messages from 24 servers (S_1–S_{24}) under one rack to the other 24 servers (R_1–R_{24}) under another rack, and none of the servers run Justitia. At the same time, we issue 8 bandwidth-sensitive applications and 1 latency-sensitive application between a pair of servers (S_{25} and R_{25}) that is controlled by Justitia. Figure 25 shows the performance with and without Justitia applied. Even in

the case where fabric congestion is out of Justitia's control, we see that Justitia can still function correctly, and Justitia still provides additional performance isolation benefits when compared with just using congestion control (DCQCN).

7   Related Work

RDMA Sharing   Recently, large-scale RDMA deployment over RoCEv2 have received wide attention [23, 41, 44, 45, 64]. However, the resulting RDMA congestion control algorithms [40, 41, 44, 64] primarily deal with Priority-based Flow Control (PFC) to provide fair sharing between bandwidth-sensitive applications inside the network. In contrast, Justitia focuses on RNIC isolation and complements them (§6.3).

Justitia is complementary to FreeFlow [35] as well. FreeFlow enables *untrusted* containers to securely preserve the performance benefits of RDMA. Because it does not change how verbs are sent to queue pairs, it can still suffer from the performance isolation problems Justitia addresses. Justitia can complement FreeFlow to provide performance isolation by implementing Justitia splitter in FreeFlow's network library and Justitia daemon in its virtual router.

SR-IOV [55] is a hardware-based I/O virtualization technique that allows multiple VMs to access the same PCIe device on the host machine. Justitia design does not interfere with SR-IOV and will still work on top of it. To provide multi-tenant fairness, Justitia can be modified to distribute credits among VMs via shared memory channel similar to [35].

LITE [62] also addresses resource sharing and isolation issues in RNICs. However, LITE does not perform well in the absence of hardware virtual lanes (Appendix C.4).

PicNIC [38] tries to provide performance isolation at the receiver-side engine congestion in software-based kernel-bypass networks, where it utilizes user-level packet processing instead of offloading packetization to an RNIC. Hence, PicNIC's CPU-based resource allocation and packet-level shaping cannot be applied to RDMA.

Swift [36] also considers receiver-side enginer congestion in software KBN by using a dedicated enging congestion window in the congestion algorithm. However, both Swift and PicNIC ignores sender-side congestion.

Offloading with SmartNICs   Recent research in Smart-NICs has focused on providing programmability and efficiency in hardware offloading [6, 19, 33, 39, 42]. However, on-NIC packet orchestration leads to tens of microsecond overhead [19, 57], making performance-related multi-tenancy support still an open problem.

NICA [18] provides isolation for FPGA-based SmartNICs by I/O channal virtualization and time-sharing of the Acceleration Functional Units. Justitia focus on normal RNICs and does not require hardware changes.

Link Sharing   Max-min fairness [9, 15, 28, 54] is the well-established solution for link sharing that achieves both sharing incentive and high utilization, but it only considers bandwidth-

sensitive applications. Latency-sensitive applications can rely on some form of prioritization for isolation [3, 25, 63].

Although DRFQ [20] deals with multiple resources, it considers cases where a packet sequentially accessed each resource, both link capacity and latency were significantly different than RDMA, and the end goal is to equalize utilization instead of performance isolation. Furthermore, implementing DRFQ required hardware changes.

Both Titan [58] and Loom [56] improve performance isolation on conventional NICs by programming on-NIC packet schedulers. However, this is not sufficient for RDMA performance isolation because it schedules only the outgoing link. Further, Justitia works on existing RNICs that are opaque and do not have programmable packet schedulers.

TAS [34] accelerates TCP stack by separating the TCP fast-path from OS kernel to handle packet processing and resource enforcement. However, TAS does not solve the type of isolation anomalies Justitia deals with. Justitia's design idea can be applied to improve isolation for TAS.

Datacenter Network Sharing   With the advent of cloud computing, the focus on link sharing has expanded to network sharing between multiple tenants [4, 8, 10, 46, 50, 53]. Almost all of them – except for static allocation – deal with bandwidth isolation and ignore predicted latency on latency-sensitive applications.

Silo [29] deals with datacenter-scale challenges in providing latency and bandwidth guarantees with burst allowances on Ethernet networks. In contrast, we focus on isolation anomalies in multi-resource RNICs between latency-, bandwidth-, and throughput-sensitive applications.

8   Concluding Remarks

We have demonstrated that RDMA's hardware-based kernel bypass mechanism has resulted in lack of multi-tenancy support, which leads to performance isolation anomalies among bandwidth-, throughput-, and latency-sensitive RDMA applications across InfiniBand, RoCEv2, and iWARP and in 10, 40, 56, and 100 Gbps networks. We presented Justitia, which uses a combination of sender-based resource mediation with receiver-side updates, Split Connection with message-level shaping, and passive machine-level latency monitoring, together with a tail latency target as a single knob to provide network sharing policies for RDMA-enabled networks.

Acknowledgments

References

[1] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *OSDI*, 2016.

[2] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, , and Michael Wei. Remote regions: a simple abstraction for remote memor. In *ATC*, 2018.

[3] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick Mckeown, Balaji Prabhakar, and Scott Shenker. pFabric: Minimal near-optimal datacenter transport. In *SIGCOMM*, 2013.

[4] Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska. End-to-end performance isolation through virtual datacenters. In *OSDI*, 2014.

[5] Apache. Apache crail. http://crail.incubator.apache.org/., 2021.

[6] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling programmable transport protocols in high-speed nics. In *NSDI*, 2020.

[7] Infiniband Trade Association. Infiniband architecture specification volume 1. https://cw.infinibandta.org/document/dl/7859, 2015.

[8] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards predictable datacenter networks. In *SIGCOMM*, 2011.

[9] J.C.R. Bennett and H. Zhang. WF^{2}Q: Worst-case fair weighted fair queueing. In *INFOCOM*, 1996.

[10] M. Chowdhury, Z. Liu, A. Ghodsi, and I. Stoica. HUG: Multi-resource fairness for correlated and elastic demands. In *NSDI*, 2016.

[11] Cloudlab. http://cloudlab.us/.

[12] RL Cruz. A calculus for network delay, Part I: Network elements in isolation. *IEEE Transactions on Information Theory*, 37(1):114–131, 1991.

[13] RL Cruz. A calculus for network delay, Part II: Network analysis. *IEEE Transactions on Information Theory*, 37(1):132–141, 1991.

[14] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, Jesse Alpert, Jing Ai, Jon Olson, Kevin DeCabooter, Marc de Kruijf, Nan Hua, Nathan Lewis, Nikhil Kasinadhuni, Riccardo Crepaldi, Srinivas Krishnan, Subbaiah Venkata, Yossi Richter, Uday Naik, , and Amin Vahdat. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *NSDI*, 2018.

[15] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *SIGCOMM*, 1989.

[16] Aleksandar Dragojevic, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. FaRM: Fast remote memory. In *NSDI*, 2014.

[17] Nick G. Duffield, Pawan Goyal, Albert Greenberg, Partho Mishra, Kadangode K Ramakrishnan, and Jacobus E van der Merwe. A flexible model for resource management in virtual private networks. In *SIGCOMM*, 1999.

[18] Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. NICA: An infrastructure for inline acceleration of network applications. In *USENIX ATC*, 2019.

[19] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, , and Albert Greenberg. Azure accelerated networking: Smartnics in the public cloud. In *NSDI*, 2018.

[20] Ali Ghodsi, Vyas Sekar, Matei Zaharia, and Ion Stoica. Multi-resource fair queueing for packet processing. 2012.

[21] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, 2011.

[22] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient memory disaggregation with Infiniswap. In *NSDI*, 2017.

[23] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. RDMA over commodity Ethernet at scale. In *SIGCOMM*, 2016.

[24] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *SIGCOMM*, 2015.

[25] Chi-Yao Hong, Matthew Caesar, and P. Brighten Godfrey. Finishing flows quickly with preemptive scheduling. In *SIGCOMM*, 2012.

[26] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure coding in Windows Azure Storage. In *USENIX ATC*, 2012.

[27] Intel. HTB Home. http://luxik.cdi.cz/~devik/qos/htb/, 2003.

[28] Jeffrey M Jaffe. Bottleneck flow control. *IEEE Transactions on Communications*, 29(7):954–962, 1981.

[29] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. Silo: Predictable message latency in the cloud. In *SIGCOMM*, 2015.

[30] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using RDMA efficiently for key-value services. In *SIGCOMM*, 2014.

[31] Anuj Kalia, Michael Kaminsky, and David G Andersen. FaSST: fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *OSDI*, 2016.

[32] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Datacenter rpcs can be general and fast. In *NSDI*, 2019.

[33] Antoine Kaufmann, Simon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. High performance packet processing with flexnic. In *ASPLOS*, 2016.

[34] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP acceleration as an OS service. In *EuroSys*, 2019.

[35] Daehyeok Kim, Tianlong Yu, Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Shachar Raindel, Chuanxiong Guo, Vyas Sekar, and Srinivasan Seshan. Freeflow: Software-based virtual RDMA networking for containerized clouds. In *NSDI*, 2019.

[36] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G Wassel, Xian Wu, Yaogong Montazeri, Behnam andand Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. Swift: Delay is simple and effective for congestion control in the datacenter. In *SIGCOMM*, 2020.

[37] Gautam Kumar, Srikanth Kandula, Peter Bodik, and Ishai Menache. Virtualizing traffic shapers for practical resource allocation. In *HotCloud*, 2013.

[38] Praveen Kumar, Nandita Dukkipati, Nathan Lewis, Yi Cui, Yaogong Wang, Chonggang Li, Valas Valancius, Adriaens Jake, Steve Gribble, Nate Foster, and Amin Vahdat. PicNIC: Predictable virtualized nic. In *SIGCOMM*, 2019.

[39] Yanfang Le, Hyunseok Chang, Sarit Mukherjee, Limin Wang Aditya Akella, Michael M. Swift, and T.V. Lakshman. Uno: Unifying host and smart nic offload for flexible packet processing. In *SoCC*, 2017.

[40] Yanfang Le, Brent Stephens, Arjun Singhvi, Aditya Akella, and Michael M. Swift. RoGUE: RDMA over generic unconverged ethernet. In *SoCC*, 2018.

[41] Yuliang LI, Harry Hongqiang Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. Hpcc: High precision congestion control. In *SIGCOMM*, 2019.

[42] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto smartnics using ipipe. In *SIGCOMM*, 2019.

[43] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kokonov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: a microkernel approach to host networking. In *SOSP*, 2019.

[44] Radhika Mittal, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. TIMELY: RTT-based congestion control for the datacenter. In *SIGCOMM*, 2015.

[45] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting network support for RDMA. In *SIGCOMM*, 2018.

[46] Jeffrey C Mogul and Lucian Popa. What we talk about when we talk about cloud network performance. *SIGCOMM CCR*, 42(5):44–48, 2012.

[47] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *USENIX ATC*, 2015.

[48] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A centralized zero-queue datacenter network. 2014.

[49] Marius Poke and Torsten Hoefler. DARE: High-performance state machine replication on rdma networks. In *HPDC*, 2015.

[50] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. FairCloud: Sharing the network in cloud computing. In *SIGCOMM*, 2012.

[51] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Carlo Contavalli, Amin Vahdat, et al. Carousel: Scalable traffic shaping at end hosts. In *SIGCOMM*, 2017.

[52] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed os for hardware resource disaggregation. In *OSDI*, 2018.

[53] Alan Shieh, Srikanth Kandula, Albert Greenberg, and Changhoon Kim. Sharing the data center network. In *NSDI*, 2011.

[54] Madhavapeddi Shreedhar and George Varghese. Efficient fair queuing using deficit round-robin. *IEEE/ACM Transactions on Networking*, 4(3):375–385, 1996.

[55] SR-IOV. Single root i/o virtualization. http://pcisig. com/specifications/iov/single_root/., 2018.

[56] Brent Stephens, Aditya Akella, and Michael Swift. Loom: Flexible and efficient nic packet scheduling. In *NSDI*, 2017.

[57] Brent Stephens, Aditya Akella, and Michael Swift. Your programmable nic should be a programmable switch. In *HotNets*, 2018.

[58] Brent Stephens, Arjun Singhvi, Aditya Akella, and Michael Swift. Titan: Fair packet scheduling for commodity multiqueue nics. In *USENIX ATC*, 2017.

[59] I. Stoica, H. Zhang, and T.S.E. Ng. A hierarchical fair service curve algorithm for link-sharing, real-time and priority service. In *SIGCOMM*, 1997.

[60] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Ana Klimovic, Adrian Schuepbach, and Bernard Metzler. Unification of temporary storage in the nodekernel architecture. In *ATC*, 2019.

[61] Mellanox Technologies. Mellanox Perftest Package. https://community.mellanox.com/docs/ DOC-2802, 2017.

[62] Shin-Yeh Tsai and Yiying Zhang. LITE kernel rdma support for datacenter applications. In *SOSP*, 2017.

[63] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better never than late: Meeting deadlines in datacenter networks. In *SIGCOMM*, 2011.

[64] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale RDMA deployments. In *SIGCOMM*, 2015.
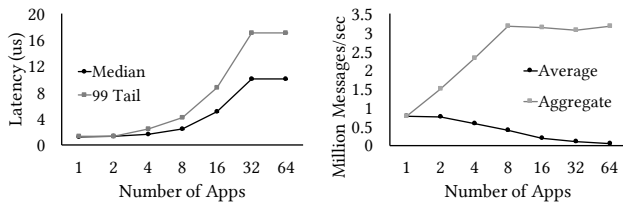
(a) *Latency* (b) *Throughput*

Figure 26: *Latencies and throughputs of multiple latency-sensitive applications in InfiniBand. Error bars (almost invisible due to close proximity) represent the applications with the lowest and highest values.*
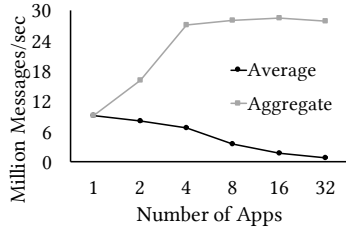
Figure 27: *Throughputs of multiple throughput-sensitive applications in InfiniBand. Error bars (almost invisible due to close proximity) represent the applications with the lowest and highest values.*
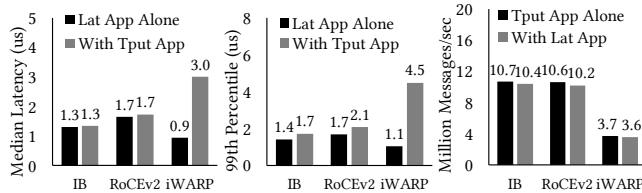
(a) *Latency App (Med)* (b) *Latency App (99th)* (c) *Throughput App*

Figure 28: *Performance anomalies of a latency-sensitive application running against a throughput-sensitive application.*

A   Hardware Testbed Summary

Table 1 summarizes the hardware we use for different RDMA protocols in our experiments.

B   Characteristics of Latency- and Throughput-Sensitive Applications in the Absence of Bandwidth-Sensitive Ones

Multiple latency-sensitive applications can coexist without affecting each other (Figure 26). Although latencies increase, everyone suffers equally. All applications experience the same throughputs as well.

Similarly, multiple throughput-sensitive applications receive almost equal throughputs when competing with each other, as shown in Figure 27.

Finally, throughput-sensitive applications do not get affected by much when competing with latency-sensitive applications (Figure 28c). Nor do latency-sensitive applications experience noticeable latency degradations in the presence of throughput-sensitive applications except for iWARP (Figure 28a and Figure 28b).
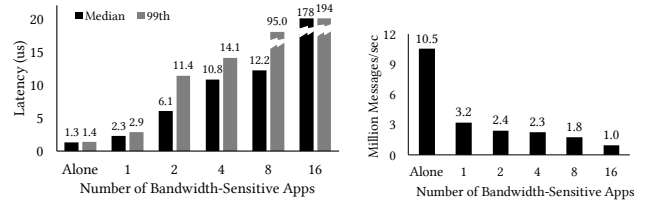
(a) *Impact on Latency* (b) *Impact on Throughput*

Figure 29: *Impact of increasing background bandwidth-sensitive applications (sending 1MB messages) in InfiniBand.*
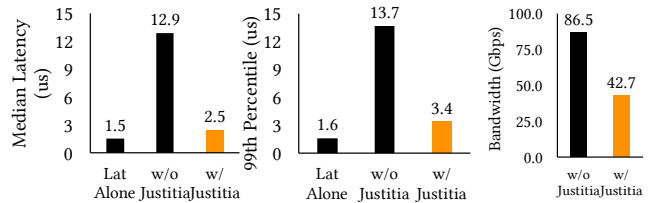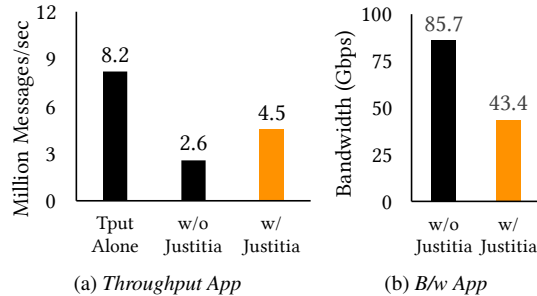

(a) *Latency-sensitive App* (b) *Bandwidth App*

Figure 30: *[100 Gbps InfiniBand] Performance isolation of a latency-sensitive application against a bandwidth-sensitive application.*

B.1   Adding More Competitors Exacerbates the Anomalies

The lack of protection for the latency-sensitive applications further exacerbates as more bandwidth-sensitive applications (or equivalently more QPs) are created. We increase the number of bandwidth-sensitive applications (each with a single QP) in our experiment to simulate more realistic datacenter applications. Although InfiniBand performs relatively well in the presence of a single background bandwidth-sensitive application (Figure 3), adding one more competitors incurs an additional drop of 2.65× and 3.79× in median and 99th percentile latencies (Figure 29a). With 16 or more bandwidth-sensitive applications, the latency-sensitive application can barely make any progress. We observed a similar trend in other RDMA technologies.

Similarly, a throughput-sensitive application loses 90% of its original throughput with 16 bandwidth-sensitive applications (Figure 29b).

Those anomalies illustrate RNIC's inability to handle multiple types of applications, which could stem from the limited number of queues inside the RNIC hardware, increasing Head-of-Line blocking of small messages.

C   Additional Evaluation Results

C.1   100 Gbps Results With/Without Justitia

Similar to the anomalies observed for 10, 40, and 56 Gbps networks (§3), Figure 30 and Figure 31 show that latency- and throughput-sensitive applications are not isolated from bandwidth-sensitive applications even in 100 Gbps networks. In these experiments, we use 5MB messages since 1MB messages are not large enough to saturate the 100 Gbps link. Justitia can effectively mitigate the challenges by enforcing performance isolation.

Protocol	NIC	Switch	NIC Capacity
InfiniBand	ConnectX-3 Pro	Mellanox SX6036G	56 Gbps
InfiniBand	ConnectX-4	Mellanox SB7770	100 Gbps
RoCEv2	ConnectX-4	Mellanox SX6018F	40 Gbps
RoCEv2 (DCQCN) (§6.3)	ConnectX-4 Lx	Dell S4048-ON	10 Gbps
iWARP	T62100-LP-CR	Mellanox SX6018F	40 Gbps
RoCEv2 (DCQCN) (§6.5 and §6.6)	ConnectX-3 Pro	HP Moonshot-45XGc	10 Gbps

Table 1: *Testbed hardware specification.*

Figure 31: *[100 Gbps InfiniBand] Performance isolation of a throughput-sensitive application against a bandwidth-sensitive application.*

C.2 Real RDMA-based Systems Require Isolation

Besides DARE, highly-optimized RDMA-based RPC systems also suffer from unmanaged RNIC resources. Here we pick two representative systems, FaSST [31] and eRPC [32], to illustrate why they require performance isolation and how Justitia effectively achieves it. To generate background traffic, we implemented a simple RDMA-based blob storage backend across 16 machines. Users read/write data to this storage using a PUT/GET interface via frontend servers. Objects larger than 1MB are divided into 1MB splits and distributed across the backend servers. This generates a stream of 1MB transfers, and the following RDMA-optimized systems have to compete with them in our experimental setup.

FaSST is an RDMA-based RPC system optimized for high message rate. We deploy FaSST in 2 nodes with message size of 32 bytes and a batch size of 8. We use 4 threads to saturate FaSST's message rate at 9.8 Mrps. In the presence of the storage application, FaSST's throughput experiences a 74% drop (Figure 32).

eRPC is an even more recent RPC system built on top of RDMA. We deploy eRPC in 2 nodes with message size of 32 bytes. We evaluate eRPC's latency and throughput using the microbenchmark provided by its authors. For the throughput experiment, we use 2 worker threads with a batch size of 8 on each node because 2 threads are enough to saturate the message rate in our 2-node setting. In the presence of the storage application, eRPC's throughput drops by 93% (Figure 33b), and its median and tail latencies increase by 67× and 40×, respectively (Figure 33a).

By applying Jusitita, FaSST's throughput improves by 2.5× (Figure 32). Justitia also improves eRPC's median (tail)
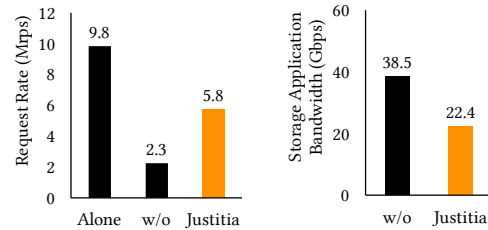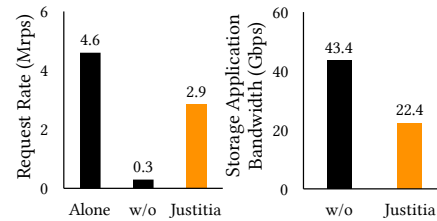

Figure 32: *Performance isolation of FaSST running against a bandwidth-sensitive storage application.*
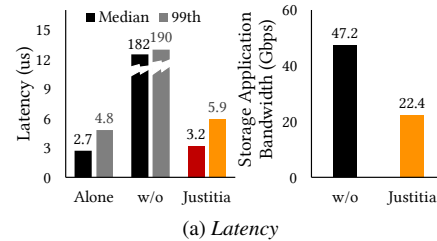
(a) *Latency*

(b) *Throughput*

Figure 33: *Performance isolation of eRPC running against a bandwidth-sensitive storage application.*

latency improves by 56.9× (32.2×) and its throughput by 9.7× (Figure 33). Note that the throughput of the storage applications drops to half of the maximum throughput in both cases because we treat the background application as a whole (and thus with equal weights to all applications, the *SafeUtil* is $\frac{1}{2}$ of the line rate), which is different from how we treat the parallel writes in the case of Apache Crial.

C.3 Handling Remote READs

RDMA READ verbs can compete with WRITEs and SENDs issued from the opposite direction (§4.5) Figure 34 shows that Justitia can isolate latency-sensitive remote READs from local bandwidth-sensitive WRITEs and vice versa.

C.4 Justitia vs. LITE

LITE [62] is a software-based RDMA implementation that adds a local indirection layer for RDMA in the Linux kernel
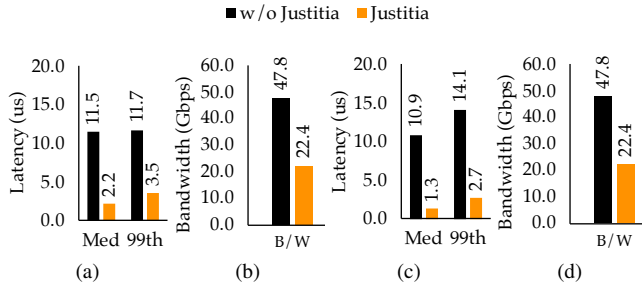
(a)   (b)   (c)   (d)

Figure 34: *[InfiniBand] (a)–(b) Justitia isolating remote latency-sensitive READs from local bandwidth-sensitive WRITEs. (c)–(d) Justitia isolating local latency-sensitive WRITEs from remote bandwidth-sensitive READs.*
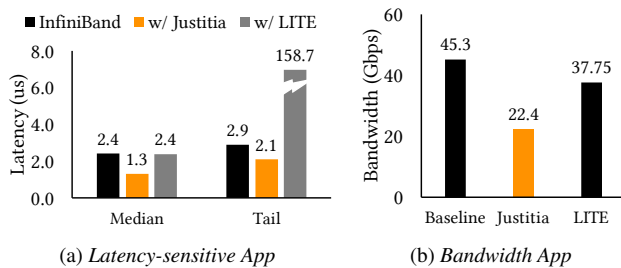

(a) *Latency-sensitive App*   (b) *Bandwidth App*

Figure 35: *[InfiniBand] Performance isolation of a latency-sensitive flow running against a 1MB background bandwidth-sensitive flow using Justitia and LITE.*
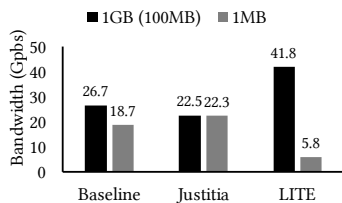

Figure 36: *[InfiniBand] Bandwidth allocations of two bandwidth-sensitive applications using Justitia and LITE. LITE uses 100MB messages instead of 1GB due to its own limitation.*

to virtualize RDMA and enable resource sharing and performance isolation. It can use hardware virtual lanes and also includes a software-based prioritization scheme.

We found that, in the absence of hardware virtual lanes, LITE does not perform well in isolating latency-sensitive flow from the bandwidth-sensitive one (Figure 35) – 122× worse 99th percentile latency than Justitia. In terms of bandwidth-sensitive applications using different message sizes, LITE performs even worse than native InfiniBand (Figure 36). Justitia outperforms LITE's software-level prioritization by being cognizant of the tradeoff between performance isolation and high utilization.

D   Sensitivity Analysis

Setting Applications Weights   To evaluate how assigning different application weights (§4.3.1) affects Justitia's performance, we launch 4 bandwidth-sensitive applications each sending 1MB message together with a latency-sensitive application, and we vary the weights of the bandwidth-sensitive applications. Figure 37 illustrates the impact of setting different application weights with latency target set to 2 μs. As

(a) *Latency*   (b) *Bandwidth*
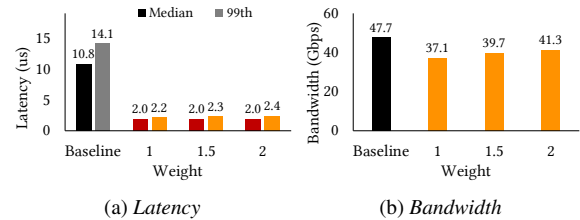
Figure 37: *[InfiniBand] Sensitivity analysis of application weights.*

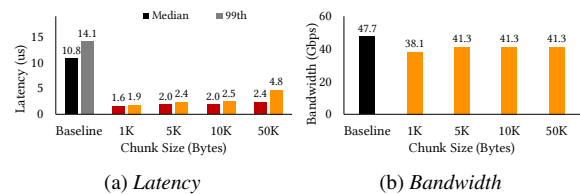(a) *Latency*   (b) *Bandwidth*

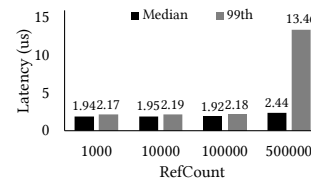**Figure 38:** *[InfiniBand] Sensitive analysis of chunk sizes.*

Figure 39: *[InfiniBand] Sensitivity analysis of RefCount.*

the weight increases, the value of *SafeUtil* increases, and thus more aggregate bandwidth share is obtained for bandwidth-sensitive applications. Higher *SafeUtil* leads to worse latency isolation, but in these experiments the effect of weights on tail latency performance is not huge. In fact, we do not find much latency performance degradation as the weight increases, illustrating the effectiveness of Justitia mitigating head-of-line blocking via its splitting mechanism. The cluster operator can choose weights based on the priority of the applications in the cluster based on the Quality-of-Service inside the cluster (similar to deciding bandwidth resources in a shared environment), or based on how much each application pays to obtain the service. Additionally, if multi-tenant fairness is desired, one can achieve that by modifying how credits are allocated in Justitia on a per-tenant basis. Justitia supports allocating tokens at multiple granularities if needed, which can be per-tenant, per-application, or per group of connections within an application.

Setting Chunk Size   When latency-sensitive applications are present, Justitia picks the smallest chunk size that still provides a wide range of bandwidth in case *SafeUtil* is high (Figure 10). Here we evaluate how setting the correct chunk size affects Justitia's performance. We use the same setting as the sensitivity analysis of application weights and set the weight to be 2. Figure 38 illustrates the experiment results with different chunk sizes. Although a smaller chunk size provides better latency isolation, it is not able to achieve *SafeUtil* and thus waste bandwidth resources. On the other hand, too big of a chunk size does not provide enough latency isolation.

Setting *RefCount* To evaluate how sensitive the value *RefCount* (§4.3.2) is, we design an experiment where initially we launch one latency-sensitive application to compete with a bandwidth-sensitive application. Once the experiment starts, we add three additional bandwidth-sensitive application, with a gap of 1 second between their arrival time. We measure the latency of latency-sensitive application after it completes 10 million messages. Figure 39 plots the results with different *RefCount* values. It turns out that Justitia tracks the tail latency closely as long as *RefCount* is not huge. In Justitia, we set the default value of *RefCount* =10000 to have some memory of latency spikes but not longer enough to impact stable performance.

E Discussion

E.1 Why not simply use hardware priority queues in the RNIC?

Mellanox NICs have priority queues, but as we mention in the paper, the number of queues they support is very limited (e.g., only 2 lossless queues in the RoCE NICs we test out), and we have illustrated such limited number priority queues are insufficient to provide isolation in Figure 23. In addition, the time needed to reconfigure and modify the mapping from applications' QPs to the priority queues is in the order of milliseconds. Last but not the least, it is sometimes also desireable to provide isolation inside a priority level (e.g., bandwidth-sensitive applications and latency-sensitive applications are both assigned with the same QoS level) where hardware priority queues will not be sufficient. Thus, using the priority queues provided by existing hardware does not solve the isolation problem that Justitia faces.

E.2 Why use only 1 QP in most of the microbenchmark experiments?

We use a small number of QPs to show that the performance isolation issues can easily occur even with a very small number of active connections. We also test with more number of QPs but the results are placed in Appendix due to limit of space. In fact, adding more QPs exacerbates the performance degradation (Figure 30 in the appendix).

E.3 How does Justitia handle the incast experiment?

Justitia leverages receiver-side updates to make sure the correct minimum rate guarantees are updated correctly at each sender. Due to large latency spike in the case of a network incast, senders will mostly like send via the minimum guaranteed rate (R_{min}) given the latency target will not be met. We discussed receiver-side updates in Section 4.5 and illustrate Justitia complements with existing congestion control and can further help reduce receiver-side engine congestion in Section 6.5.

E.4 Does reference flow and receiver-side updates create additonal congestion in a large scale deployment?

The reference flow sends small messages (10 Bytes every 20 μs) and only amount to a very small Gbps number (1e6 / 20 * 10 / 1e9 * 8 = 0.004 Gbps), which consumes less than 0.1% of the total link capacity even at nodes with only 10 Gbps link, and thus is not likely to generate any hot spot in the network. When the server broadcasts the receiver-side update, the message is sent using SEND and RECV with a message size of 16 Bytes. With even 1000 client machines this amounts to around 16KB total message size, which is too small to create a potential congestion problem.

In the case of a large-scale latency-sensitive flow incast, if congestion indeed happens, DCQCN will work together with Justitia since it is the major congestion control dealing with fabric congestion. In this scenario, adding more latency-sensitive flows does not prevent Justitia guaranteeing bandwidth share of bandwidth hungry applications.

In the current design of Justitia, the bandwidth-sensitive applications can be rate-limited due to a coexisting latency-sensitive application which is launched at the same host but sends data to a different destination. This is intended behavior to mitigate the anomalies caused by contention at sender-side RNICs, which happens regardless of whether two competing applications are targeting the same receiver. We defer a comprehensive fabric-level solution which involves multiple senders and receives as our future work.

E.5 How to ensure all cooperating SW uses the right protocols and protocol versions?

To deploy Justitia, one only needs to install the Justitia Daemon code and a modified Mellanox driver code on the host machine, and Justitia is compatible with all existing RDMA protocols, including RDMA over Infiniband and RoCE. In datacenter deployments, cluster management tools like Ansible can be used to ensure the appropriate code is deployed at each machine. Additionally, it is straightforward to upgrade Justitia. Because each server in Justitia operates independently, it is not necessary for the same version of Justitia to be deployed across the cluster. Justitia will operate as long as servers are running some version of Justitia.

E.6 How can Justitia be implemented in hardware?

Without having a software layer to split the large RDMA operations before they arrive at the NIC, one probably need to somehow control how the NIC issues PCIe reads. Hardware is often optimized for performance, which in fact is why we are having such isolation issues, so simply decreasing the size of each PCIe reads will definitely affect its maximum throughput performance. To bring Justitia into the hardware design, similar to what we have done in the software layer, the hardware need to recognize when splitting and pacing is needed to provide isolation, and when it should process at

maximum capacity for higher utilization.

E.7 Long-term value of Justitia

As RNICs keep evolving, its performance isolation issues may be mitigated in newer hardware designs. The purpose of this work is to show that there exist such isolation issues in current kernel-bypass networks and illustrate one working approach to mitigate the issue. Design ideas presented in this work can inform hardware designers when developing future RNIC as well as programmable NIC designs.

F Future Research Directions

Interesting short-term improvements of this work include, among others, dynamically determining an application's performance requirements to handle multi-modal applications, handling idle applications, and extending to more complicated application- and/or tenant-level isolation isolation policies. Long-term future directions include implementing Justitia logic on an RNIC and integrating Justitia with congestion control algorithms.

We highlight these immediate next-steps in the following:

Dynamic Classification (Strategyproof Justitia). Applications may not always correctly or truthfully identify their flow types. To improve Justitia, it is possible to modify the driver to monitor QP usage and automatically identify whether individual connections are bandwidth-, throughput-, or latency-sensitive. This would provide support for multi-model applications.

Idle Applications. It is straightforward to support idle applications in Justitia without wasting bandwidth. If a bandwidth hungry app stops sending messages for a long time but does not exit, the driver and daemon can work together to stop token issuing when tokens are not being used and a configurable backlog of tokens has been accumulated.

Justitia at Application and Tenant Levels. Currently, Justitia isolates applications/tenants by treating all flows from the same originator as one logical flow with a single type. However, for an application with flows with different requirements or for a tenant running multiple applications competing with another tenant only running a single application, more complex policies may be desirable. With Justitia, it is straightforward to instead support per-tenant, per-application, or per-flow-group isolation. This is done by allocating tokens at multiple different granularities.

Co-Designing with Congestion Control. Although Justitia effectively complements DCQCN (§6.3) in simple scenarios, DCQCN considers only bandwidth-sensitive flows. A key future work would be a ground-up co-design of Justitia with DCQCN [64] or TIMELY [44] to handle all three traffic types for the entire fabric with sender- and receiver-side contentions (§6.5). While network calculus and service curves [12, 13, 29, 59] dealt with point-to-point bandwidth- and latency-sensitive flows, their straightforward usage can be limited by multi-resource RNICs and throughput-sensitive flows. At the fabric level, exploring a Fastpass-style centralized solution [48] can be another future work.

NetHint: White-Box Networking for Multi-Tenant Data Centers

Jingrong Chen   Hong Zhang[†]   Wei Zhang   Liang Luo[#]   Jeffrey Chase   Ion Stoica[†]   Danyang Zhuo

Duke University [†]*UC Berkeley* [#]*University of Washington*

Abstract

A cloud provider today provides its network resources to its tenants as a black box, such that cloud tenants have little knowledge of the underlying network characteristics. Meanwhile, data-intensive applications have increasingly migrated to the cloud, and these applications have both the ability and the incentive to adapt their data transfer schedules based on the cloud network characteristics. We find that the black-box networking abstraction and the adaptiveness of data-intensive applications together create a mismatch, leading to sub-optimal application performance.

This paper explores a white-box approach to resolving this mismatch. We propose NetHint, an interactive mechanism between a cloud tenant and a cloud provider to jointly enhance application performance. With NetHint, the provider provides a *hint* — an *indirect indication* of the underlying network characteristics (e.g., link-layer network topologies for a tenant's virtual machines, number of co-locating tenants, network bandwidth utilization), and the tenant's applications then adapt their transfer schedules accordingly. The NetHint design provides abundant network information for cloud tenants to compute their optimal transfer schedules, while introducing little overhead for the cloud provider to collect and expose this information. Evaluation results show that NetHint improves the average performance of allreduce completion time, broadcast completion time, and MapReduce shuffle completion time by 2.7×, 1.5×, and 1.2×, respectively.

1 Introduction

Data-intensive applications (e.g., network functions, data analytics, deep learning) have increasingly moved to the cloud for resource elasticity, performance, security, and ease of management. The performance of the cloud network is critical for these applications' performance. Cloud providers have thus spent significant effort to optimize various aspects of cloud networks, including network topology [34, 73, 76], congestion control and network stack [3, 33, 42, 44, 69, 77, 92], load balancing [2, 46, 63, 88], bandwidth guarantee [6, 9, 43, 48, 51, 67], debugging [7, 31], fault recovery [53], hardware [8, 27, 52, 58], and virtualization [66].

Today, a cloud provider exposes the network to its tenants as a black box: the cloud tenants have little visibility into their expected network performance (e.g., a constant worst-case bandwidth assurance) or the underlying network characteristics including the link-layer network topology, number of co-locating tenants, and instantaneous available bandwidth.
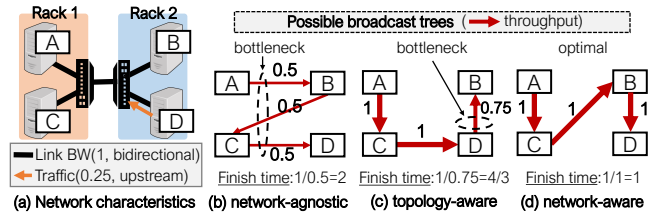

Figure 1: **Applications have the ability and the incentive to adapt their transfer schedules based on network characteristics:** Consider broadcasting a unit-size data object from VM A to VM B, C, and D. (a) shows the network characteristics, all links have bidirectional bandwidth of 1. VM D has upstream background traffic of 0.25. (b) to (d) show possible broadcast trees and their corresponding broadcast finish time. The arrows represent traffic flows and the numbers represent the throughput.

The black-box model has worked well for decades due to its simplicity. However, with the emergence of popular data-intensive applications (e.g., data analytics, distributed deep learning, and distributed reinforcement learning) in the cloud, we observe that such a black-box model is no longer efficient (§2). The crux is that many of these emerging applications have both the *ability* and the *incentive* to adapt their transfer schedules based on the underlying network characteristics, but it is difficult to do so with a black-box network.

Consider broadcast, an important communication primitive in reinforcement learning and ensemble model serving. Figure 1 shows an example that VM A broadcasts to VM B to VM D. Figure 1b shows a possible broadcast tree constructed under the black-box model. Without the underlying network characteristics, the broadcast tree is *network-agnostic*, which introduces link stress on the cross-rack link. Figure 1c shows a broadcast tree based on the topology information (i.e., topology-aware), which improves the broadcast finish time from 2 to $\frac{4}{3}$ time units by minimizing the cross-rack traffic. Figure 1d shows a broadcast tree based on both the topology and bandwidth information (i.e., network-aware). It builds an optimal broadcast tree that avoids the congested upstream link on VM D, further improving the finish time to 1 time unit. The performance gains increase for data center networks that have larger oversubscription ratios or more skewed traffic.

The above example illustrates a fundamental *mismatch* between the black-box nature of existing network abstractions and the ability of a data-intensive application to adapt its traffic. With the black-box model, the cloud tenant is unaware of the network characteristics, and the cloud provider is unaware of the application communication semantics and the transfer schedule. This misses an opportunity for the cloud tenants and

the cloud provider to adapt the data flows to the underlying network topology and conditions to enhance performance and efficiency for these applications. The potential gains are substantial: our benchmark experiment on AWS shows that the allreduce latency for a deep learning experiment varies by up to 2.8× across different allreduce transfer schedules. One candidate approach is for applications to probe and profile the network and then plan their data flows accordingly [5, 57]. A second option is to report their possible transfer schedules to the provider for the provider to choose. We observe that these alternatives introduce substantial communication latency and system overhead (§2.2).

In this paper, we explore a *white-box* approach to resolve this mismatch. One possibility would be for the cloud provider to expose the physical network topology, the VM locations, along with bandwidth assurances to the application. However, this approach has two major drawbacks. First, exposing VM placement and data center network topology may compromise security for cloud tenants and can raise concerns for the cloud provider (§2). Second, the bandwidth available to a tenant depends on the communication patterns of other tenants, which may be highly dynamic. Predictions that are not timely or not accurate may do more harm than good.

This paper explores an alternative approach. We design and implement NetHint, a mechanism for a cloud tenant and cloud provider to interact to enhance the application performance jointly. The key idea is that the provider provides a *hint* — an *indirect indication* of the bandwidth allocation to a cloud tenant (e.g., a virtual link-layer network topology, number of co-locating tenants, network bandwidth utilization). The tenant applications then adapt their transfer schedules based on the hints, which may change over time. NetHint balances confidentiality and expressiveness: on one hand, the hint avoids exposing the physical network topology or traffic characteristics of other tenants (§9). On the other hand, we show that the hint provides sufficient network information to enable tenants to plan efficient transfer schedules. (§5).

The effectiveness of NetHint relies on addressing three important challenges. First, *what information should the hint contain?* We provide each cloud tenant with a virtual link-layer network topology along with available bandwidth on each link in the virtual topology. This allows applications to adapt their transfer schedules to avoid network congestion.

The second challenge is *how to provide this hint at a low cost*. We design a two-layer aggregation method to collect network statistics on the hosts. We designate a NetHint server in a rack to aggregate network characteristics in the rack. NetHint servers then use all-to-all communication to exchange network characteristics globally. A cloud tenant can thus query its rack-local NetHint server for hints.

The final challenge is *how should applications react to the hint*. We present several use cases for NetHint to optimize communication in a range of popular data-intensive applications including deep learning, MapReduce, and
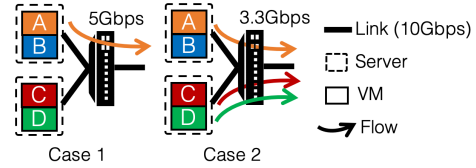

Figure 2: **Examples to illustrate the black-box networking abstraction: tenants cannot predict their network performance.** VM A to D are placed in two servers. All links have 10 Gbps bandwidth. We assume bandwidth is statically partitioned on the end host (each VM can get at most 5 Gbps).

serving ensemble models. The takeaway is that for all these applications, tenants can use the NetHint information via simple scheduling algorithms. Adaptation also has a downside: hints can be stale and adapting transfer schedules based on stale information can hurt performance. We design a policy for applications to adapts flexibly with different hints in different scenarios: applications use temporal bandwidth information when background network conditions are stable and adaptation overhead is low, and otherwise applications fall back to using only the time-invariant topology information (§6).

We evaluate the overheads and the potential performance gain of having NetHint in data centers using a small testbed and large-scale simulations. Our results show that NetHint speeds up the average performance of allreduce completion time in distributed data-parallel deep learning, broadcast completion time in ensemble model serving, and MapReduce shuffle completion time in distributed data analytics by 2.7×, 1.5×, and 1.2×, respectively. Moreover, these benefits are cheap to obtain: NetHint incurs modest CPU, memory, and network bandwidth overheads.

In summary, this paper makes the following contributions:

- We identify a mismatch between the current black-box network abstraction and the communication needs of data-intensive applications.
- We explore a white-box networking approach for multi-tenant data centers.
- We design and implement NetHint, a low-cost system to allow data-intensive applications to adapt their data transfer schedules to enhance performance.

2 Background

2.1 Black-Box Networking Abstraction

Today, the networking abstraction a cloud has is merely a per-VM bandwidth allocation at the end hosts. The abstraction is a *black box*: tenants are unaware of the underlying network characteristics including network topology, number of co-locating tenants, and instantaneous available bandwidth. As a result, the cloud tenants cannot predict their network performance. Figure 2 shows an example. Even with a static allocation of 5 Gbps per VM, VM A cannot predict its network performance because it depends on the traffic demand of other VMs. VM A can get only a bandwidth of 3.33 Gbps when
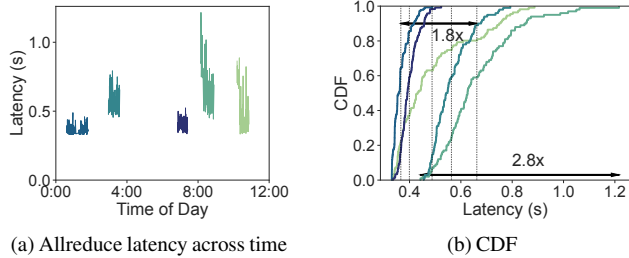
(a) Allreduce latency across time   (b) CDF

Figure 3: Empirical allreduce (256MB) latency of 5 trials. Two trials may have different VM allocations spatially, and each trial contains 100 consecutive runs. (a) shows 5 trials over different times of a day. In (b), each line is the latency CDF of a trial. Each vertical line is the mean latency for a trial. Allreduce latencies vary both across time (up to 2.8x) and across VM allocations (up to 1.8x).

two flows of VM C and D cause congestion inside the network (case 2). Even with work-conserving bandwidth guarantees, a VM's network performance depends on other VMs.

To quantify this effect, we benchmark allreduce latency on Amazon Web Service (AWS). Allreduce is a collective communication primitive that is commonly used for distributed deep learning. It aggregates a vector (i.e., gradient updates in deep learning) across all worker processes (each running in its own VM). In our experiment, we launch 32 g4dn.2XL (with Linux kernel 5.3) instances in the EC2 US-East-1 region and test ring-allreduce latency with NVIDIA NCCL (version 2.4.8)—the most popular collective communication library for deep learning—for 100 consecutive runs. We repeat the above experiment for 5 trials, and different trials may have different VM placements on the physical topology. Figure 3 shows our findings: ring-allreduce performance on 256MB buffer varies both spatially across different trials and temporally within a trial. Comparing across different trials, the fastest trial has a $1.8\times$ better mean performance than the slowest trial; comparing the 100 runs within a trial, the fastest run is up to $2.8\times$ faster than the slowest run.

2.2 Adaptiveness in Data-Intensive Applications

Besides reinforcement learning and ensemble model serving, which can broadcast model and input data adaptively, as illustrated in Figure 1, we show that many other applications also have both the ability and incentive to adapt their transfer schedules based on the underlying network characteristics.

Many distributed data analytics workloads contain network-intensive shuffle phases between different job stages. For example, the shuffle in MapReduce applications creates an all to all data transfer between the map and reduce stages. The shuffle phase accounts for a large portion of the execution time for many data analytics workloads [16], and numerous studies [4, 15, 16, 39, 84, 87, 90] have demonstrated that optimizing shuffle performance significantly improves application performance. Given network characteristics, distributed data analytics applications can change their transfer schedules (by changing the task placement) to minimize shuffle completion time. Figure 4a
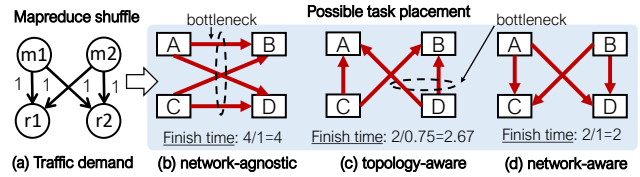

Figure 4: **MapReduce jobs can adapt transfer schedules via task placement.** Assume the same network characteristics as in Figure 1a. (a) shows the traffic demand for a MapReduce shuffle. Each arrow represents a unit traffic. (b) to (d) show possible task placement and the corresponding shuffle finish time.

shows the shuffle traffic for a MapReduce job with two mappers (m1 and m2) and two reducers (r1 and r2). We observe from Figure 4b to Figure 4d that allocating mappers and reducers based on the topology and bandwidth information effectively improves this shuffle completion time from 4 to 2 units. Moreover, emerging task-based distributed systems (e.g., Ray, Dask, Hydro) support applications with dynamic task graphs. Similar to the MapReduce example, we can change the transfer schedule of these applications by choosing different VMs to place a task.

Moreover, many deep learning jobs are network-intensive. This claim is validated by numerous recent studies [14, 35, 40, 71, 86] and observations from production clusters (e.g., Microsoft [30, 41, 82] and ByteDance [65]). In particular, as mentioned in §2.1, deep learning jobs contain an allreduce phase to synchronize gradient updates among workers in each training iteration. As shown in Figure 5, an allreduce phase has multiple candidate topologies. For example, the allreduce traffic can be sent via a ring connecting all the workers with a flexible ordering (Figure 5a and Figure 5b). Or, we can build an allreduce tree to (1) aggregate gradient updates to one of the workers, and (2) send the aggregated gradient updates back in the reverse direction (Figure 5c and Figure 5d). Different allreduce topologies introduce different transfer schedules. Thus, given network characteristics, deep learning jobs can change their transfer schedules by selecting the algorithm and configuration of allreduce.

2.3 Addressing the Mismatch

The black-box nature of the existing networking abstraction and the adaptiveness of data-intensive applications create a mismatch. Data-intensive applications would benefit from more network information from the cloud provider to configure their transfer schedules, but black-box networking hides this information.

Solutions based on the black-box abstraction. There are two approaches to address this mismatch without modifying the existing black-box networking abstraction. One possible approach is to let the cloud provider optimize the communication for tenants as a cloud service. To this end, we first have to develop a general networking API for cloud tenants to express their communication semantics, traffic loads and optimization objectives to the cloud provider. The API design should be similar to the coflow abstraction [16] or the virtual cluster ab-
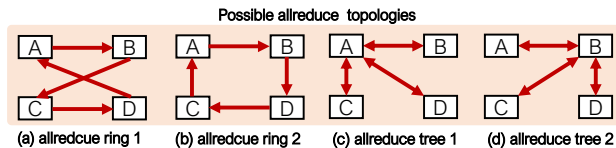
Possible allreduce topologies

Figure 5: **Allreduce can be performed with different topologies**. (a) to (d) show 4 possible allreduce topologies to perform allreduce among the 4 VMs (workers).

straction [9], but more general to support a large variety of possible traffic patterns and user-defined objectives. Moreover, a recent measurement study [78] shows that major public clouds exhibit high bandwidth variability at a time granularity of seconds. Thus it is hard, if not impossible, for the cloud provider to perform timely network scheduling for thousands of tenants in a centralized manner, while ensuring network SLAs (e.g., defined via the networking API) for each tenant respectively.

Another potential approach is for cloud tenants to run extensive performance profiling in their allocated VMs [29, 49, 57]. For example, PLink [57] probes the VM pair-wise bandwidth and latency with DPDK and uses K-means clustering to reverse engineer the underlying network topology. This allows it to achieve high allreduce performance by choosing a good allreduce algorithm. Choreo [49] uses 3-step measurements to pinpoint congested links in the data center network to schedule data analytics workloads. Similar approaches were explored decades ago on Internet traffic routing on wide-area overlay networks [5]: picking a high-performance Internet path based on user measurement. Unfortunately, this approach is both costly, as each tenant/user has to profile the network independently, and slow, because the probing phase delays the start of the application. The PLink authors told us that they use 10000 packets to determinte bandwidth between a pair of hosts. Choreo generates 3 minutes of probe traffic to infer the network characteristics for 10 VMs.

A white-box network abstraction? Given the deficiencies of the two black-box based approaches, we instead explore a white-box approach: the provider reveals essential information about the network characteristics to the tenant, and the tenants then optimize their transfer schedules accordingly.

One possible way to achieve this objective is for the cloud provider to reveal to a tenant the location of each VM in the physical link-layer network topology, and estimate available bandwidth between each of the VM-pairs. However, this method can raise security and competitive issues. First, exposing VM allocations in the physical network introduces privacy risks for cloud tenants. For example, a malicious user can locate a targeted tenant's VMs and perform attacks. Second, the exposed VM allocation information can raise competitive concerns for the cloud provider. For instance, this information might be valuable for competitors to learn a cloud provider's scheduling policies, thus, lowering its competitive advantage. Third, the bandwidth a tenant can acquire depends on the transfer schedules of *all* the tenants, and a single change in transfer schedule of one tenant may trigger a recalculation
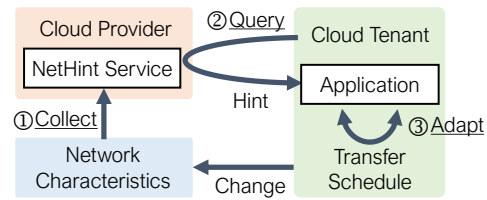

Figure 6: NetHint overview. NetHint service collects network characteristics. Cloud tenants poll hints from NetHint service and adapt their transfer schedules.

for all the tenants. As such, it is computationally expensive for the cloud provider to update the bandwidth shares in real time. Moreover, an application's bandwidth also depends on *its own* transfer schedule. For example, in Case 2 of Figure 2, if VM A sends one extra flow, the total egress bandwidth of VM A increases to 5 Gbps[1]. As a result, without knowing a tenant's transfer schedule, the cloud provider cannot provide accurate bandwidth estimates to its tenants.

3 NetHint Overview

NetHint is an interactive mechanism between a cloud tenant and a cloud provider to jointly enhance the application performance. The key idea is that the provider provides a *hint* — an *indirect indication* of the underlying network characteristics (e.g., a virtual link-layer topology for a tenant's VMs, number of co-located tenants, network bandwidth utilization) to a cloud tenant. As illustrated in Figure 6, the provider provides a NetHint service, which periodically (100 ms by default) collects the hint information to capture changes of the underlying network characteristics. A tenant application can query the NetHint service to get the hint information, and then adapt its transfer schedules based on this provided hint. Note that NetHint does not change the fairness mechanism of the underlying network. A tenant can opt in/out any time — whether or not to use NetHint will *not* affect its fair share of the network.

The hint provides a white-box network abstraction which includes additional network information to tenants. As such, users can infer their best transfer schedule without substantial probing latency or communication overhead with the provider. The hint exposes neither the physical network topology nor the location of a tenant's VMs within it (e.g., which racks). Compared with providing bandwidth information, the hint relieves the provider from the burden of calculating accurate bandwidth allocations. Moreover, compared with calculating bandwidth allocation, it is easier to acquire accurate hint messages (e.g., a virtual link-layer topology for a tenant's VMs, number of co-located tenants, network bandwidth utilization). As such, the provider is free from the potential risk of providing inaccurate information.

We require NetHint to be: (1) *readily deployable:* all the mechanisms are implementable using commodity hardware; (2) *low cost:* the cloud provider can collect network characteristics with minimal CPU, memory, and bandwidth

---

[1] Assume per-flow fair sharing in the network.

overheads; (3) *useful:* data-intensive workloads can leverage the hints to achieve high performance. To achieve these goals, NetHint's design and implementation must address three questions. First, what hints should be provided to the tenants? Second, how should cloud providers collect the hints with low cost? Third, how should applications use the hints to adapt their transfer schedules?

NetHint describes a virtual link-layer topology that connects a tenant's VMs. In addition, NetHint provides to the tenant recent utilization summaries and counts of co-locating tenant connections on shared network links in the virtual topology. This information allows the tenant to adapt its transfer schedules based on both the topological and temporal hot spots in the network. Further, our design ensures that the only additional information NetHint exposes is aggregated network statistics across all tenants. It is thus difficult for a tenant to acquire information about any individual other tenant. (§4.1)

For the second question, our preferred approach to collecting hints is to measure network traffic in the physical switches using network telemetry, e.g., sketching [54, 55]. However, sketches depend on specific programmable switch features, which are not widely deployed. Instead, our prototype employs a host-driven approach, in which each machine monitors local flows and transmits flow-level statistics to a NetHint measurement plane. One machine in each rack runs a *NetHint server* process to aggregate the rack-level information. These NetHint servers exchange information using periodic all-to-all communication. A cloud tenant connects to the local NetHint server to fetch hints. We show that this approach allows NetHint to provide timely hints to tenants with low CPU and bandwidth overheads (§4.2).

As for the third question, we consider two aspects of adaptation in response to the hints. First, we observe that the adaptation algorithm should take into account the application transfer schedule and semantics to maximize the performance gain. To this end, we consider several use cases for NetHint which cover a range of popular data-intensive applications, including (1) choosing allreduce algorithms in distributed deep learning, (2) constructing broadcast trees for serving ensemble models, and (3) placing tasks in MapReduce frameworks. For each case, we show how applications can adapt their transfer schedules based on the information in the hint. The takeaway is that for all of these examples, tenants can make use of the NetHint information via simple scheduling algorithms (§5).

Second, we explore the drawbacks of adaptation: it introduces extra computational overhead, and may be ineffective or even harmful or unstable if network conditions change too rapidly. We conclude that the adaptation algorithm should use different sets of hints depending on network changing frequency and adaptation overhead. For example, we find that if an application has a non-negligible latency to collect hints and compute the transfer schedules, the bandwidth information may be stale and thus may negatively affect the application performance (detailed in §6). Based on this intuition, we design

Notations & Descriptions	
$\mathcal{T}$	A virtual topology connecting all the tenant's VMs
l	A virtual link in virtual topology $\mathcal{T}$
B_e^l	A tenant's bandwidth share on link l
B_t^l	Total bandwidth on link l
B_r^l	Residual bandwidth on link l
n^l	Number of shared objects on link l

Table 1: Notations and descriptions for NetHint.

a policy for applications to react to hints in a flexible manner: under stable network conditions and low adaptation overheads, applications use both bandwidth and topology information to maximize the performance gain of adaptation. Otherwise, applications use only the stable topology information (§6).

4 Providing NetHint Service

4.1 What Is in the Hint?

NetHint exposes a virtual link-layer topology $\mathcal{T}$ to a cloud tenant. The tenant's virtual topology abstracts the network as a tree data structure in which the tenant's VMs are leaf nodes. A link in the tree represents one or more physical links in the data center network, and an interior node may abstract a region of switches and links. The prototype uses a three-layer tree that captures how VMs are distributed among racks in a data center and collapses the network structure above the rack level into a single root node. VMs residing in the same rack are in the same subtree. The virtual topology abstraction does not reveal racks or servers where the tenant has no presence. Following the common observation that congestion losses often occur at the rack level [12, 43, 60, 89], these virtual topologies in the NetHint prototype ignores congestion at any structure above the rack level [23]. It is possible to represent more structure by adding layers to the tree. The tree approximation presumes that the data center network is able to balance its load, so that traffic among children of an abstract node see similar available bandwidth. There is a rich literature on efficient network load balancing for data centers [2,21,22,26,28,36,46,47,63,88], and some of them are readily deployable with commodity hardware.

NetHint allows applications to react to temporal hot spots in the network. For this purpose, NetHint exposes an estimate of utilization on each virtual link l. Recall Case 1 in Figure 2, now assume the orange flow from VM A uses only 2 out of 10 Gbps. If the tenant of VM B knows the network utilization information, it can infer that VM B can send traffic at 8 Gbps. As such, NetHint provides (1) the total bandwidth B_t^l and (2) the residual bandwidth B_r^l on each virtual link l. However, we find that this information alone is insufficient for an application to adapt its transfer schedule, especially when links are congested. For example, even if one link l has already reached 100% utilization, a tenant can still send flows through l and get a fair bandwidth share.

Shared objects and fairness models. In fact, the bandwidth share depends on the fairness model implemented by the cloud provider. Per-flow-fairness and per-VM-pair-fairness

are enforced naturally for RDMA-based networks because modern RDMA NICs can be configured to choose either of them. Per-flow-fairness is ensured for containerized clouds because cloud users cannot modify the kernel TCP stack. For traditional TCP-based and VM-based clouds, many recent studies [18, 37, 62] describe how to enforce per-VM-pair-fairness. With the increasing programmability of modern switches, it now becomes possible to implement other fairness models in the network [74, 83], such as per-tenant fairness.

Consider an application placing 3 connections on a 100 Gbps network link with 7 existing connections from 3 other tenants. We assume each flow can reach 100 Gbps throughput. With per-flow fairness model, the application should get 30 Gbps bandwidth. With per-tenant fairness model, the application should get 25 Gbps bandwidth.

The example indicates that the bandwidth share also depends on the number of *shared objects* on each link l. The definition of shared object depends on the fairness model: it is a flow (VM-pair, tenant) under per-flow (per-VM-pair, per-tenant) fairness, respectively.

To provide bandwidth information, NetHint exposes the number of shared objects n^l on each link l. Taken together, NetHint provides a tuple (n^l, B_t^l, B_r^l), which includes both the current link utilization and the number of shared objects.

Bandwidth estimation. The information in the virtual topology enables a tenant to estimate its available bandwidth on each virtual link l efficiently. More formally, consider a tenant who plans to place k^l shared objects on link l in its transfer schedule. If link l is an in-network link in virtual topology $\mathcal{T}$ (i.e., not attached to any VM), the bandwidth share the tenant gets can be estimated as:

$$B_e^l = \max\left(\frac{k^l}{n^l + k^l}B_t^l, B_r^l\right) \tag{1}$$

Equation 1 indicates that when the link is under-utilized, the tenant can use up all the residual bandwidth B_r^l, and even if the link is already congested, the tenant can at least achieve its fair share based on the number of shared objects.

If link l is an edge link (i.e., attached to one VM), the bandwidth share is also affected by the underlying sharing approach. More specifically, denote the per-VM bandwidth guarantee provided by the sharing approaches as B_v, we have:

$$B_e^l = \begin{cases} \min(B_v, \max(\frac{k^l}{n^l+k^l}B_t^l, B_r^l)) & \text{static partitioning} \\ \max(\frac{k^l}{n^l+k^l}B_t^l, B_r^l, B_v) & \text{work-conserving} \end{cases} \tag{2}$$

Sources and impact of inaccuracy We acknowledge that both Equation 1 and Equation 2 are approximations and can sometimes be inaccurate. First, some shared objects (i.e., tenant, VM-pair, or connection) may have traffic demands less than their fair network share, thus calculating the exact value of B_e^l requires knowing the traffic demand for each shared object. NetHint does not provide per-object information, as doing so introduces security concerns and significant overhead given the huge number of such objects. Second, since a virtual

link corresponds to the aggregation of multiple parallel paths in the physical topology, the estimation may be inaccurate under poor network load balancing across these parallel paths. We note that this is less likely to happen with recently proposed data center network load balancing designs.

Despite these inaccuracies in bandwidth estimation, our results (§8) show that even the three-level tree approximation is sufficient to adapt the transfer schedules and improve the performance of our target applications. Moreover, evaluation results also show that the benefits degrade gracefully with the quality of the approximations.

Alleviating security and competitive issues. Compared with a naive white-box solution that exposes VM allocation information and physical network topology, NetHint has alleviated the security and the competitive concerns. First, NetHint does not expose the physical location of allocated VMs, so a tenant cannot learn the provider's VM allocation policy. Second, our network statistics are aggregated over all other tenants, so it is difficult for a tenant to infer from them the network behavior of any other individual tenant. Finally, network topology among a tenant's VMs is already accessible even in today's black-box model via user probing approaches, e.g., as presented in PLink [57] and Choreo [49]. NetHint does provide easier access to this information, but we believe this does not increase the security risks. Note that NetHint does not fully eliminate these issues, and we discuss them in §9.

4.2 Timely NetHint with Low Cost

User query overhead The virtual topology is presented as a set of links (each with a Link ID). Each virtual link has its associated B_t. The temporal utilization information for each link includes a tuple of three fields (Link ID, n, B_r). Each field occupies 8 bytes. As such, the amount of data returned by a query is small. Consider a cloud tenant that has rented 100 VMs allocated across 10 racks. As upstream and downstream virtual links are considered separately, the number of virtual links equals twice the sum of the number of VMs and the number of racks the tenant occupies. The amount of query information thus has $(100 + 10) \times 2 \times 3 \times 8 = 5280$ Bytes.

There is no value or incentive for a tenant to query at a higher frequency than the information update period of NetHint (100 ms by default). Tenant VMs communicate with a NetHint server through TCP connections with rate limits that prevent queries more frequent than once per 50 ms.

Collection overhead We design a two-layer host-driven aggregation approach to collect timely hint information with low cost. Recall that we select one machine in each rack to run a NetHint server process. Each machine collects flow-level network characteristics from its operating system, and sends them to its rack-local NetHint server periodically. The information each machine has to send to the local NetHint server is a virtual link ID plus one (n, B_r) for each virtual machine to ToR link and another (n, B_r) containing only the traffic transmitting

across the rack, for adding its contribution to the ToR uplink's (n, B_r). Each field is 8 bytes, so the total data size per virtual link is $(1+2\times2)\times8 = 40$ bytes. It is necessary to consider the upstream and downstream bandwidth independently, so each virtual machine or ToR has two associated virtual links. For example, assuming a physical machine has 10 VMs, it sends $40\times2\times10 = 800$ bytes of data to the NetHint server in each period. We set the information update period to 100 ms by default. Thus, the total aggregated information for one NetHint server is two (n, B_r) for every VM-to-ToR virtual link and the ToR uplink in the virtual topology. The NetHint servers then use all-to-all communication to exchange their aggregated information.

Suppose a data center has 1000 racks, and every rack has 20 machines. In each information update period, a local NetHint server gathers 16 KB information (800 bytes × 20 machines). With a 100 ms update period, the total amount of cross-rack traffic introduced by the all-to-all information exchange is $16\,\text{MB}/100\,\text{ms} = 1.3$ Gbps per rack. Let's assume each rack has outgoing bandwidth of 500 Gbps. Then the bandwidth overhead of NetHint is 0.26%.

Failure detection and recovery   NetHint is a best-effort service, and applications should be prepared to function without hints, e.g., if their rack-local NetHint servers become unavailable due to failures such as link failure and server crashing. In this case, applications just revert the transfer schedule to a default one assuming no known network characteristics until a new NetHint server is available in the rack.

5   Adapting Transfer Schedules with NetHint

We find that most data-intensive applications can be categorized into two classes, based on how they can adapt to network characteristics. For each application class, we show that adapting transfer schedules corresponds to an optimization problem. Our goal here is not to present the optimal algorithm to solve the scheduling problems. Rather, our goal is to show that a broad set of distributed applications can benefit from NetHint using simple scheduling algorithms.

5.1   Optimizing Collective Communication

Many data-intensive applications run a high-level collective communication primitive (e.g., broadcast, allreduce) among a set of processes. Any such operation can be accomplished flexibly via a large set of possible *overlay topologies* among all the processes. For example, a broadcast can be performed with different broadcast trees connecting all the receivers, and an allreduce may employ different allreduce topologies (e.g., tree-allreduce or ring-allreduce). For all these communication primitives, the choice of overlay topologies affects only the efficiency (i.e., finish time) but not the correctness. Many popular ML applications belong to this category:

- **Data-parallel deep learning:** each server holds a replica of the model and calculates gradients locally. Servers use allreduce to synchronize gradients in each training iteration.

- **Reinforcement learning:** the trainer process in reinforcement learning repeatedly broadcasts the model (i.e., policy) to a dynamic set of agents.
- **Serving ensemble models:** multiple servers run DNN models simultaneously to predict the label on the same input data, and then use voting to decide the final output. For every input data batch, the front-end server broadcasts it to a set of servers holding different DNNs.

Moreover, as the object of collective communication is usually a vector of numbers, we can partition the object and apply different overlay topologies on each partition. For example, a broadcast can be accomplished via multiple broadcast trees, with each broadcast tree transferring a different (weighted) portion of the broadcast object. Similarly, an allreduce can be performed via a weighed combination of different allreduce topologies. The transfer schedule thus depends on both the choices of overlay topologies and their corresponding weights.

With NetHint, the tenant can estimate the bandwidth B_e^l available on each link l based on Equation 1 and Equation 2. For a transfer schedule s, denote the volume it transfers on each link l as d_s^l. The corresponding latency of the schedule can be estimated as $\max_l(d_s^l/B_e^l)$. Thus, we have:

Problem statement: *Given the virtual topology $\mathcal{T}$ and the estimated bandwidth on each virtual link l, find a transfer schedule that minimizes the latency $\max_l(d_s^l/B_e^l)$.*

To solve the above problem, one major challenge is that the number of candidate transfer schedules can be huge. For example, there can be $O(n^{(n-2)})$ possible broadcast trees to broadcast a message to n processes [79]. One possible solution is to use tree packing algorithms [13,25,79]. However, since the goal here is to show the usefulness of NetHint information rather than to find the optimal algorithm, we design simple heuristics to solve the problem. We first sample a random set of overlay topologies (broadcast and allreduce trees) which cross each rack only once. We then use linear programming to find the best weight assignment among these trees, so that the transfer schedule minimizes the latency $\max_l(d_s^l/B_e^l)$.

5.2   Optimizing Task Placement

Many distributed applications execute based on a task graph describing the tasks and their dependencies. The task graph can be static (i.e., task graph is known before the workload runs) [19,85] or dynamic (i.e, tasks arrive as the workload runs) [61]. Since different tasks may send and receive different amounts of data, the placement of tasks onto VMs determines the transfer schedule among the VMs. Applications in data analytics frameworks and task-based distributed systems therefore can benefit from network-aware task placement:

- **Data analytics frameworks** [32,85]: data analytics workloads contain network-intensive shuffle phases between different job stages. One shuffle phase creates an all-to-all communication between a set of sender tasks and receiver tasks, so task placement controls the shuffle performance.

Notations & Descriptions	
T_b	Average changing period of the background network condition
T_u	Duration of a transfer schedule being used
T_a	Latency to adapt (collecting information and computing a schedule)
T_s	Staleness of the hint
p	A threshold defined by the ratio between total adapting latency and JCT

Table 2: Important factors related to the impact of staleness.

- **Task-based distributed systems** [38, 61] are increasingly popular in industry. In these applications, the task graph is dynamic and generated at runtime. Tasks launch after fetching input objects from upstream tasks. As such, efficient task placement can minimize the task launch latency reducing the object fetch time.

Problem formulation   For both applications, we can formulate the task placement as a classical network embedding problem. Denote the set of tasks as $\mathbb{T}$ and the set of VMs as $\mathbb{V}$. Compared with the problem statement in §5.1, which selects an efficient data transfer schedule, here we need to find an embedding $\mathcal{E} : \mathbb{T} \mapsto \mathbb{V}$ given the transfer schedule among all tasks. The algorithm inputs and optimization goals are the same as the problem statement in §5.1, except that the latency is calculated as $\max_l(d_e^l/B_e^l)$. d_e^l is the transfer volume on link l introduced by embedding $\mathcal{E}$.

We make minor modifications to the greedy heuristics proposed in Hedera [1] to solve the embedding problem. We first sort all tasks in $\mathbb{T}$ based on the amount of data they receive in decreasing order (no need if $|\mathbb{T}| = 1$). We then place tasks one by one following this order. When placing a task to $\mathbb{V}$, we optimize greedily for the objectives described in the problem statement. Before processing the next task, we update the cross rack traffic and d_e^l based on the placement.

6   Flexible Adaptation for Stale Information

Staleness of NetHint information   The staleness of NetHint information during job execution is affected by the following two factors (notations listed in Table 2). First, an application controller can have a non-negligible latency to collect hints and compute the transfer schedules based on the hints, which makes the hints stale when being applied. We denote the adaptation latency as T_a.

Second, applications can adapt to hints periodically. For each adaptation period, the schedule calculated based on the previous hint will be used for the entire duration T_u. Note that for recursive jobs (e.g., model serving), recomputing the schedule for every iteration introduces too much latency. To this end, we fetch hints and recompute the schedule every k iterations, so that the latency to compute transfer schedule is within a portion p (e.g., 10% by default) of the job execution time. Moreover, for jobs that adapt the task placement based on hints (e.g., MapReduce), the adaptation period T_u equals job completion time, as the task placement usually cannot be changed during job execution.

Taken together, the staleness of NetHint information is quantified as $T_s = T_a + T_u$, which is the combination of both above factors. T_a is the total latency of four steps. The first three steps are to collect hints: sending host network characteristics to NetHint service, NetHint service exchanges rack-level network characteristics, and applications querying the NetHint service. The maximum latency for these three steps combined is 300 ms (100 ms per step due to NetHint frequency), so we use 150 ms as the estimate for the average case latency. The last step is to compute the transfer schedule, and it is application-specific (Figure 8). In our evaluation, a deep learning job of 64 workers requires 10 ms to compute its transfer schedule. We thus set $T_a = 150 + 10 = 160$ ms. We set $T_u = 100$ ms to keep the compute overhead to be less than 10% of the total running time.

Impact of the stale information   The impact of stale information depends on the relative relationship between (1) the staleness of the information; and (2) the stability of the underlying network condition. Assume the background network condition changes every T_b time in average. A hint with staleness T_s much less than T_b can still be helpful since the network condition is likely to be similar with the condition T_s time ago. In contrast, a hint with staleness T_s much larger than T_b will be misleading, since the current network condition may be very different from the condition T_s time ago. In this case, adaptation with misleading hints can negatively affect the application performance (Figure 12d).

Flexible adaptation based on application and network condition.   There are two takeaways from the above analysis. First, stale information should not be used when it is misleading. Regarding this, one approach is to simply ignore the provided hints and run applications as we run them today. However, as we show in motivating examples (e.g., Figure 1c and Figure 4c), the link-layer network topology alone can be useful for some types of applications to reduce the amount of cross-rack traffic. Compared with the bandwidth information, topology information is more stable and not affected by network dynamics.

Therefore, we propose NetHint-TO, a class of scheduling algorithms that use only the stable topology information from NetHint. For example, with NetHint-TO, we create a ring that crosses each rack only once for ring-allreduce and a chain that crosses each rack only once for tree-broadcast.

The second takeaway is that there is no one-size-fits-all solution. Each application should have two scheduling algorithms, one uses bandwidth information (in §5) and another one uses stable topology information only (NetHint-TO). We design a policy to choose between these two algorithms based on both the application and the network conditions (i.e., T_b, T_u, T_a). More specifically, when $T_s < T_b$, applications use the scheduling algorithm in §5 to calculate the optimal schedule based on both bandwidth and topology information. When $T_s \geq T_b$, applications adopt NetHint-TO to minimize the impact of stale information.

7 Implementation

We implement NetHint using 4600 lines of Rust code. 2300 additional lines of code are in NetHint server to provide NetHint to cloud tenants. The algorithms for applications to adapt transfer schedules (i.e., MapReduce, allreduce, and broadcast) are implemented using 149, 216, and 144 lines of code. We use `lpsolve` [56] for solving linear programs.

To compute the hints in our testbed, we take an endhost-based approach. We hook an eBPF program into the OS kernel. The eBPF program counts the total number of bytes going within the rack and outside the rack. A userspace program polls the counters from the eBPF program every 10 ms and maintains a moving average of the number of existing shared objects (i.e., flows, in a per-flow fairness model). The userspace program sends the number of shared objects and traffic data to the NetHint server every 100 ms. In a deployment environment where SmartNICs is available, we can also program the SmartNICs to implement this logic.

NetHint server binds to a TCP port, where VMs connect to to fetch hints. NetHint server uses a single thread to respond to NetHint queries. A single thread is enough for our design because queries are not frequent.

For an application to use NetHint, we need to modify the application. For traditional collective communication, the transfer schedule is static and decided before runtime. Recent collective communication designs have shown that transfer schedules can be dynamically decided at runtime [93]. NetHint can help these dynamic collective communication designs to decide on an efficient transfer schedule based on network characteristics. These dynamic collective communication designs can query and adapt transfer schedule every k iterations before issuing data transfer operation. For task placement, the global scheduler of a distributed system (e.g., master in MapReduce [19]) queries the NetHint server and uses both the task information and the NetHint information to decide task placement. For our evaluation purpose, we build a dynamic scheduler for collective communication and a task scheduler for MapReduce tasks according to the descriptions above.

8 Evaluation

8.1 Setup and Workloads

We evaluate NetHint using an on-premise testbed and large-scale simulations. Our setting is that hosts ensure work-conserving bandwidth guarantee for VMs and the network ensures per-flow fairness. We compare NetHint with the scenarios where cloud tenants (1) do not consider network characteristics and (2) probe the network to reverse-engineer the network characteristics and then adapt transfer schedules. For user probing, we assume network information is always correctly reverse engineered. We assume the probing strategy is the following: For a tenant that owns n hosts, user probing runs in $n/2$ rounds, where each round's latency is either the latency to send 10000 packets or 1 second, whichever is smaller, to measure

throughput and latency between $n/2$ pairs of hosts. [2] Similar to NetHint, user probing adopts the same strategy to periodically update the transfer schedule, but with a lower frequency due to its higher overheads. We calculate user probing's frequency using the same method described in the second paragraph of §6.

We use a mix of two types of background traffic to simulate skewed and long-tailed traffic in data centers [3, 12, 70, 89]. One slow-moving background traffic occupies 0-50% bandwidth of the link capacity on each link in a Zipfian distribution. The slow-moving background traffic occupies 10% bandwidth in total and changes every 10 seconds. The other is a fast-moving background traffic which is on all links and occupies 0-10% bandwidth of the link capacity in a uniform random fashion. The fast changing background traffic changes every 10 ms. We use the following workloads. We run each experiment 5 times and report the average speedup for each job. To quantify the overall speedup, we also measure the arithmetic average of speedups across jobs.

Distributed data-parallel deep learning. We test the allreduce completion time. The job sizes are either 16 or 32 (in terms of number of nodes) with equal probability. For each allreduce job, we set the buffer size to be 100 MB ($\approx$ the size of ResNet-50). We run 100 jobs and assume jobs arrive as a Poisson process. We choose Poisson lambda = 24 seconds, so that the average network utilization approximates to 12%.

Serving an ensemble of ML models. We test the broadcast completion time. We use the same job size distribution described in Hoplite [93]. We run 100 jobs and assume jobs arrive as a Poisson process. We choose Poisson lambda = 8 seconds, so that the average network utilization approximates to 12%.

MapReduce. We test the latency of the data shuffling phase of MapReduce. We use Facebook's MapReduce trace [17], which contains 500 MapReduce jobs and their arrival time. We assume the traffic is divided evenly from a reducer to the mappers.

8.2 NetHint in Testbed Experiments

We build a 6-server testbed. Each server has a 100 Gbps Mellanox ConnectX-5 NIC and two Intel 10-core Xeon Gold 5215 CPUs (2.5 GHz). These machines are connected via an emulated 40 Gbps 2-stage FatTree network using a single 100 Gbps Mellanox SN2100 switch through self-wiring. 3 machines are in one rack, and the rest 3 machines are in the other rack. The oversubscription ratio on our network is 3. Each machine runs 4 VMs where each VM is guaranteed 10 Gbps through fair-queuing on the NICs.

Overheads. We already provide analysis of bandwidth overheads in §4.2. Now the remaining question is how much overhead NetHint incurs in terms of latency and CPU cycles.

---

[2]We believe this is a best-case scenario for existing user probing techniques. Plink [57] sends 10000 packets per VM-pair to reverse engineer link-layer topologies. Choreo [49] uses a 3-step strategy to pinpoint congested links and its first step is measure pair-wise bandwidth. It takes 3 minutes to reverse engineer the network conditions for 10 VMs (90 VM-pairs).
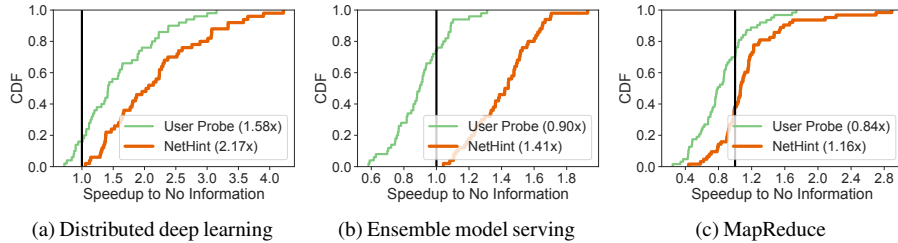
(a) Distributed deep learning   (b) Ensemble model serving   (c) MapReduce

Figure 7: **Testbed results:** NetHint's speedup on testbed for allreduce in data-parallel distributed training, broadcast in ensemble ML model serving, and mapreduce shuffle compared with user probing and not using network information. Numbers in the legend shows the average of speedups compared with running applications without network information.
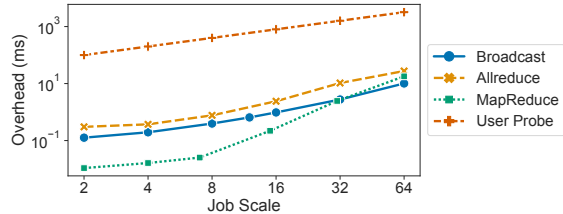

Figure 8: **Testbed results:** Latency to compute transfer schedules.

# Racks	CPU Util. (%)	Memory (MB)	Latency (ms)
6	0.06	4.53	10.60
24	0.14	5.90	10.73
96	0.41	19.28	11.91
240	0.66	78.16	13.73

Table 3: **Testbed results:** The system overhead of a NetHint server in CPU utilization, memory, and information collection latency.

Collecting statistics from eBPF program is instant, and the polling period for flow statistics is 10 ms.

To measure the overheads in large deployment, we use each CPU core in our testbed to emulate a rack by instantiating a NetHint server per-core. We use pidstat to measure the CPU cycles and memory footprint on NetHint server. Table 3 shows the result. When the number of racks scale up to 240 racks, the CPU time spent on NetHint servers is negligible, i.e., less than 0.66%. The memory footprint on each NetHint server is small (less than 80 MB) and scales with the number of racks mainly due to the increase in the hint size. The latency to collect network information is less than 14 ms.

We implement the algorithms described in §5. We test the computation latency of running each algorithm at different scales (number of workers). Figure 8 presents the results. The latency to make a scheduling decision remains low, ranging from 10 us to 30 ms. Compared with the computation latency, the extra latency introduced by user probing is much higher, ranging from 100 ms to 3 seconds. The round-trip latency to fetch hints takes 100 us because it is rack-local.

Results. NetHint improves application performance. Figure 7 shows the normalized speedup to running applications without network information. Using user probing speeds up the communication by 1.6x for distributed data-parallel deep learning and slows down the communication by 1.1x and 1.2x for serving an ensemble of ML models, and MapReduce shuffle, respectively. NetHint speeds up communication of these workloads by 2.2x, 1.4x, and 1.2x, substantially outperforming user probing. NetHint can outperform user probing because collecting hints is more lightweight than each application individually probing the network characteristics. User probing hurts many ensemble model serving and MapReduce jobs because of the probing overheads. In addition, we notice that a small portion of jobs in Figure 7c are penalized. On our testbed, the job log shows

that there are on average 2.8 jobs sharing the rack bandwidth. One job arrival or departure changes the network condition for all the other jobs on the rack. However, the task placement decision cannot be changed during job execution, and thus the initial placement can be imperfect. In contrast, deep learning and model serving workloads in Figure 7 do not severely suffer from this problem, as they can timely modify the transfer schedule for each iteration based on the latest NetHint information.

8.3 NetHint in Simulations

We use simulations to evaluate NetHint in large-scale deployments and in various operating environments. Our simulator is written in 5000 lines of Rust. The simulation is at flow level, and throughput of each flow is the result of solving a max-min fairness formula based on traffic demand. We simulate a CPU cluster and a GPU cluster individually. Both the CPU and GPU clusters have 150 racks. In the GPU cluster network, each rack has 6 machines with 100 Gbps NIC, and each rack has total upstream bandwidth of 200 Gbps. In the CPU cluster network, each rack has 18 machines with 100 Gbps NIC and the total upstream bandwidth is 600 Gbps. The oversubscription ratios are both 3. In the CPU cluster, each machine has 4 VMs. In the GPU cluster, each machine only has 1 VM. All VMs have bandwidth guarantee of 25 Gbps.

Results. Figure 9 shows the NetHint's speedup of the three workloads in our simulations. In summary, the trend of the simulation results matches what we have observed on the testbed. NetHint speeds up communication by 2.7x, 1.5x, and 1.2x, respectively. On allreduce, the speedup is higher than that on the testbed because the number of hosts involved in a job is larger than that on the testbed, and thus the amount of cross-rack traffic is also larger, giving NetHint more room to optimize transfer schedules.

User probing incurs substantial overheads in both traffic and latency. Figure 10 shows the overheads of using NetHint and
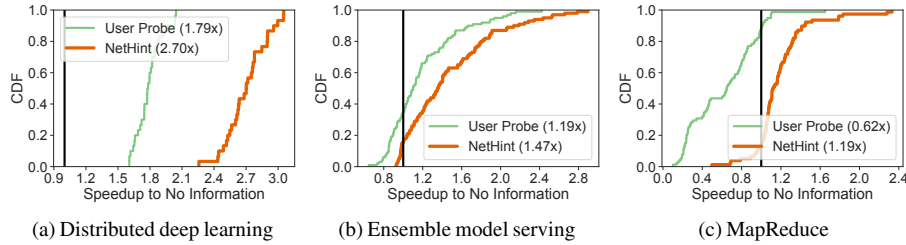
(a) Distributed deep learning (b) Ensemble model serving (c) MapReduce

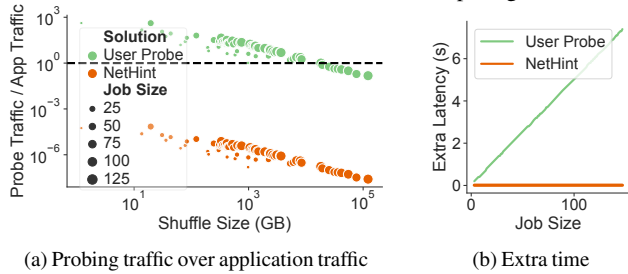Figure 9: **Simulation results:** Comparing NetHint with dynamic user probe in the default background traffic setting.

(a) Probing traffic over application traffic (b) Extra time

Figure 10: **Simulation results [MapReduce]:** Extra overhead for MapReduce jobs comparing NetHint and user probing.

(a) Coarse-grained workloads (b) Non-negligible overhead to compute a transfer schedule

Figure 11: **Simulation results [Model serving]:** Using topology information alone can outperform using bandwidth information.

user probing in MapReduce. The amount of overhead depends on both MapReduce shuffle size and job size. Figure 10a shows the extra traffic introduced by NetHint and user probing over application traffic. NetHint only adds less than 0.1% extra traffic. User probing, in contrast, adds 15% to 420% extra traffic, and 90% of jobs double their traffic. This is because user probing needs to generate probe traffic, and each application has to probe independently. For large shuffle sizes, the probing traffic is less of a concern because it constitutes a smaller fraction of the total traffic. Figure 10b shows the extra latency due to probing and fetching hints for MapReduce jobs of various sizes. NetHint only adds a constant RTT-level extra latency which is negligible. User probing has a large latency overhead, which is linear in job size. This is expected because user probing needs to run for $n/2$ rounds, where n is the job size. There are a set of MapReduce jobs that are penalized substantially by user probing (as shown in Figure 9c). These are MapReduce jobs with large job sizes but with small shuffle sizes.

When should NetHint use topology information only? As we have described in §6, there are two situations we prefer letting NetHint use topology information only: (1) workload granularity is large, and (2) overhead of computing a transfer schedule is non-negligible. To demonstrate these situations, we set the slow-moving background traffic change frequency to every 0.2 seconds. Other environment settings remain the same as those in previous simulations.

To show the case when background traffic changes faster than job completion time, we run 100 broadcast jobs with the model sizes increased to 1 GB. We let NetHint recompute a new broadcast strategy every iteration (but we still guarantee that the computational overhead is under a certain threshold $p = 10\%$). We use NetHint-TO to denote using only topology information when calculating the transfer schedule. We use NetHint-BW to denote using bandwidth information when
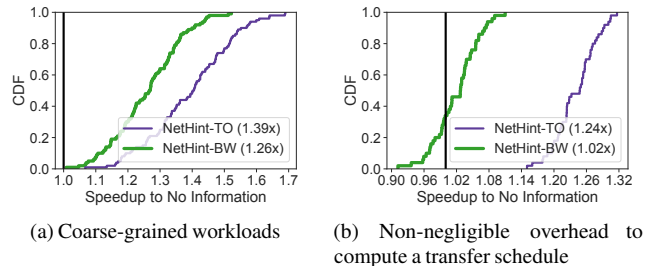
calculating the transfer schedule. Figure 11a shows that NetHint-TO and NetHint-BW speed up the communication by 1.4x and 1.3x. NetHint-BW is slightly slower than NetHint-TO. Applying a bandwidth-aware algorithm does not bring benefit compared with using topology information only because the background traffic changes even within a single broadcast. Instead, it can slow down the job due to the additional overhead to compute data transfer schedules.

To demonstrate an extreme example for the computational overhead, we run 100 broadcasts of 64 workers with data size set to 12 MB, and we double the bandwidth capacity of ToR switch. Figure 11b shows that NetHint-TO and NetHint-BW speed up by 1.2x and 1.0x compared with no information. NetHint-BW cannot improve because the computation latency using LP is large in contrast to the broadcast latency on such a small data size. It has to adapt its traffic less frequently ($\approx 0.2s$) to ensure the compute overhead is within 10% of the total job completion time. Without being affected by inaccurate hints, NetHint-TO aims to minimize the cross-rack traffic, thus achieving better performance.

Figure 12 shows which adaptation method NetHint choose under different background traffic change periods and oversubscription ratios. The result demonstrates that NetHint chooses the best of NetHint-TO and NetHint-BW for all the three applications we use and also for both oversubscription ratio of 3 and 1.5.

Inaccurate bandwidth estimation. The bandwidth estimations in Equation 1 and Equation 2 is based on approximations, as the accurate estimation requires knowing the traffic demand for each tenant. One question to ask is whether NetHint's design fundamentally relies on the accuracy of bandwidth estimation. To answer this question, we intentionally add noise to the input of NetHint. Having additional noise of x% means the link utilization provided to NetHint is between 100-x%

(a) Deep Learning (Oversub=3)   (b) Deep Learning (Oversub=1.5)   (c) Model Serving (Oversub=3)   (d) MapReduce (Oversub=3)
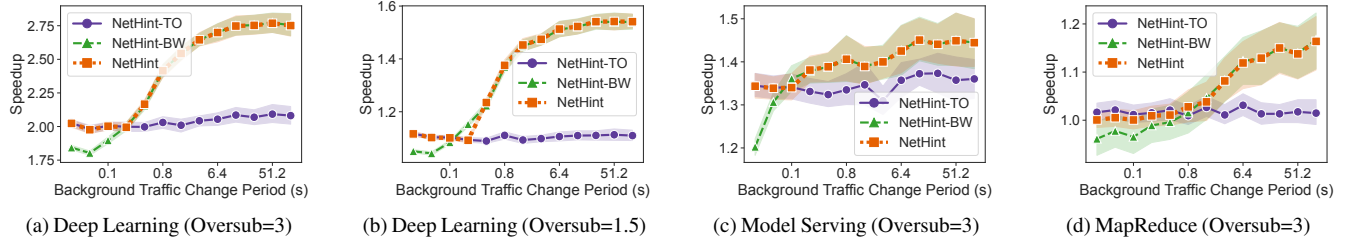
Figure 12: **Simulation results:** Average speedup to background traffic change period under two different topology settings. The shaded area represents 95% confidence interval.
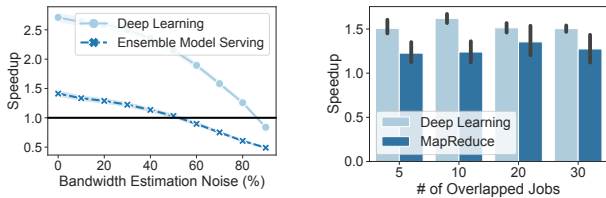

Figure 13: **Simulation results:** NetHint's speedup to not using network information when we add noise to the input of NetHint.

Figure 14: **Simulation results:** NetHint's performance when varying the number of overlapped jobs.

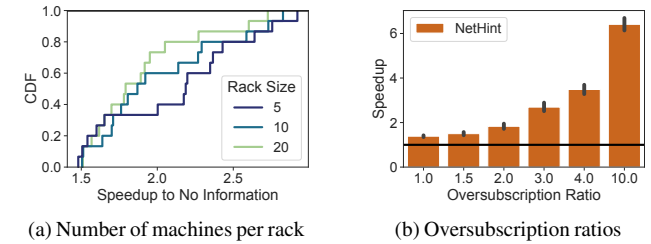(a) Number of machines per rack   (b) Oversubscription ratios

Figure 15: **Simulation results [Distributed deep learning]:** NetHint's speedup to not using network information when we evaluate under different deployment environments.

and 100+x% of the actual utilization. We then evaluate the speedup of allreduce and broadcast jobs. Figure 13 shows the result. NetHint's speed up degrades gracefully. NetHint can still outperform not using network information when there is up to at most 50% noise.

Performance stability. To evaluate if NetHint's performance remains stable when the number of NetHint users is large, we increase the number of overlapped jobs. For deep learning, we enlarge the rack size to allow more jobs to share a ToR link and start all the jobs at the beginning. For MapReduce, we scale up the job arrival rate to create more overlapping among jobs. Figure 14 shows that NetHint can constantly achieve performance gain over not using network information.

Sensitivity to network configurations. We evaluate NetHint's speedup under different network configurations in terms of the number of machines per rack and oversubscription ratios. We vary the number of machines per rack while keeping the oversubsription the same at 3. Figure 15a shows that NetHint can reduce the communication latency consistently for different rack sizes. We then vary the oversubscription ratio. Figure 15b shows that NetHint's improvement compared with not using network information increases as oversubscription ratio increases. This is because, when oversubscription ratio is high, the cross-rack communication is more likely to become the bottleneck. NetHint can mitigate this bottleneck by reducing the total amount of cross-rack traffic.

Performance gain over perfect user probing. In our evaluation, for n hosts, user probing is performed in $n/2$ rounds. In each round, it measures the bidirectional bandwidth and latency between $n/2$ pairs of hosts in parallel for a certain duration (default to 100 ms). Moreover, we show some evidence that it can be difficult to design better user probing

technique to achieve similar performance as NetHint. First, we demonstrate how low the user probing duration has to be in order to achieve similar performance as NetHint. For this, we artificially reduce the probing duration while ensuring probing is accurate in simulations. Figure 16a shows the result: even when probing duration is reduced to 1 ms, NetHint still has a small performance advantage over user probing. Second, we show that such a low probing duration (i.e., 1 ms) for accurate bandwidth estimation can be difficult due to data center microbursts. We simulate data center microbursts based on measurement results in Facebook data centers [89] and calculate whether probing for x ms is sufficient to predict the average bandwidth of 100 ms. Figure 16b shows that if we measure for less than 25 ms, there is a 50% probability that the estimation error is above 75%. This is because there are gaps between microbursts, when a busy link is temporarily idle. Probing for such a short amount of time may not detect any traffic.

Does NetHint work for other fairness models? The rapid advancement in the programmability in emerging programmable switches makes it possible to implement other types of fairness models in the network [74, 83]. This trend makes it interesting to also understand NetHint's potential performance gains if we move to other fairness models in the future. We simulate the same allreduce jobs except that we modify our simulator for different fairness models. As shown in Figure 17, the trend of the simulation results matches what we have obtained in a per-flow based fairness setting.

9 Discussion

Herd behaviors. Tenants adapting transfer schedules with provided hints in a distributed way can potentially cause stability issues. For example, given the information of an under-utilized

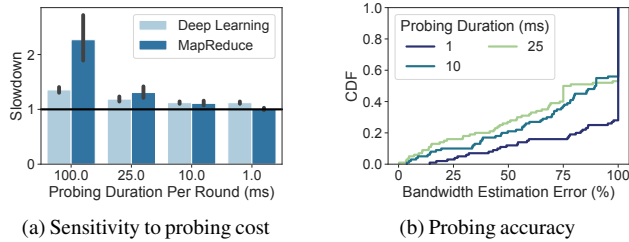(a) Sensitivity to probing cost   (b) Probing accuracy

Figure 16: **Simulation results:** The speedup of user probing to NetHint and the relative bandwidth estimation difference under different assumptions of probing durations. The black line in (a) represents NetHint.

link, many tenants may make identical choices to move traffic to this link, causing congestion. Such herd behavior causes load imbalance and performance oscillation in distributed load balancing problems [2,59,88]. We note that herd behavior is a common problem in some specific applications such as distributed load balancers. There are also standard techniques such as adding random jitters, and power of two choices to alleviated herd effect [59]. Whether and how NetHint should help specific applications avoid herd behavior is an interesting future direction. In the workload and setting of our evaluation, NetHint's speedup does not decrease when we increase the number of overlapped jobs (Figure 14). This infers that the performance of NetHint is not significantly affected by herd behavior.

Other competitive concerns for NetHint. NetHint exposes network utilization information to tenants. Network utilization can be a sensitive information. For example, one can infer whether a cloud has customers and whether a cloud provider does a decent job in network load balancing. NetHint makes it easy for a customer to compare network characteristics at different times. If a customer finds that the achievable bandwidth is reducing via NetHint, there may be a risk that the customer will switch to another cloud provider.

10 Related Work

Sharing network bandwidth. How to share network among many applications or cloud tenants is one of the oldest problems in computer networks. Today, network sharing is opaque to the application or cloud tenants. Within a single tenant, bandwidth sharing is through the fairness property of the underlying congestion control algorithms [24]. Across tenants, a cloud provider usually enforces strong isolation through static bandwidth allocation [68] or work-conserving bandwidth guarantee [9,10,50] on the NICs. It is difficult to enable either static bandwidth allocation or work-conserving bandwidth guarantee in the network because commodity switches have limited numbers of hardware queues. NetHint is complementary to these bandwidth sharing design: NetHint does not change any fairness property of the network. NetHint provides guidance for applications to use the network bandwidth better. A non-participating tenant can simply ignore the hint.

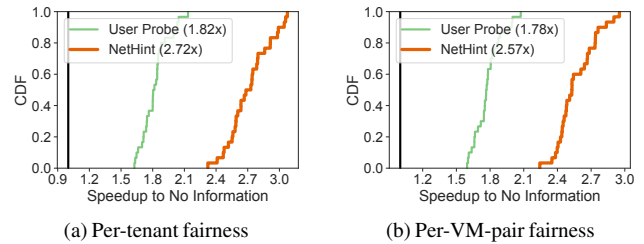**Collective communication and task placement based on**

(a) Per-tenant fairness   (b) Per-VM-pair fairness

Figure 17: **Simulation results [Distributed deep learning]:** Speedup for other fairness models.

network characteristics. Many related works optimize collective communication [20, 29, 45, 64, 79] or task placement [39, 49, 75, 80, 91] based on topology or bandwidth information. Similar considerations can also be applied inside OS for multi-core machines [11]. Most of these solutions assume the network topology or bandwidth information is already known. As such, NetHint can work in complementary with these solutions by providing them timely network information. Second, these works do not consider a multi-tenant environment. They assume workloads can be controlled by a logically centralized controller, while we assume each tenant's workload is controlled only by the tenant itself. Because tenants do not know other tenants' communication patterns, this knowledge needs to be provided either through cloud provider's support as proposed in this paper or using probing.

User probing. In addition to PLink and Choreo, many past works [5, 72, 81] also propose to measure network characteristics in wide-area networks to choose Internet route. NetHint is different in two aspects: (1) NetHint does not rely on active probe, and thus NetHint has low cost. NetHint simply reads counters directly from NICs or operating systems. (2) NetHint is for distributed applications that can adapt their transfer schedules rather than choosing routes in the network.

11 Conclusion

Today, the networking abstraction a cloud tenant has is a black box. This prevents a tenant's data-intensive applications from adapting the data transfer schedules to achieve high performance. We design and implement NetHint, a new paradigm for division of work between a cloud provider and its tenants. A cloud provider provides a hint, network characteristics (e.g., a virtual link-layer network topology, number of co-locating tenants, available bandwidth), directly to its tenants. Applications then adapt their transfer schedules based on these hints. We demonstrate the performance gain of NetHint on three use cases of NetHint including allreduce communication in distributed deep learning, broadcast in serving ensemble models, and scheduling tasks in MapReduce frameworks. Our evaluations show that NetHint improves the performance of these workloads by up to 2.7×, 1.5×, and 1.2×, respectively. Our source code is available at https://github.com/crazyboycjr/nethint.

Acknowledgement

References

[1] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI*, 2010.

[2] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. CONGA: Distributed Congestion-Aware Load Balancing for Datacenters. In *SIGCOMM*, 2014.

[3] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP (DCTCP). In *SIGCOMM*, 2010.

[4] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *OSDI*, 2010.

[5] David Andersen, Hari Balakrishnan, Frans Kaashoek, and Robert Morris. Resilient Overlay Networks. In *SOSP*, 2001.

[6] Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska. End-to-end Performance Isolation Through Virtual Datacenters. In *OSDI*, 2014.

[7] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang (Harry) Liu, Jitu Padhye, Boon Thau Loo, and Geoff Outhred. 007: Democratically Finding the Cause of Packet Drops. In *NSDI*, 2018.

[8] Hitesh Ballani, Paolo Costa, Raphael Behrendt, Daniel Cletheroe, Istvan Haller, Krzysztof Jozwik, Fotini Karinou, Sophie Lange, Kai Shi, Benn Thomsen, and Hugh Williams. Sirius: A Flat Datacenter Network with Nanosecond Optical Switching. In *SIGCOMM*, 2020.

[9] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards Predictable Datacenter Networks. In *SIGCOMM*, 2011.

[10] Hitesh Ballani, Keon Jang, Thomas Karagiannis, Changhoon Kim, Dinan Gunawardena, and Greg O'Shea. Chatty Tenants and the Cloud Network Sharing Problem. In *NSDI*, 2013.

[11] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *SOSP*, 2009.

[12] Theophilus Benson, Aditya Akella, and David A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *IMC*, 2010.

[13] Chandra Chekuri and Kent Quanrud. Near-linear time approximation schemes for some implicit fractional packing problems. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 801–820. SIAM, 2017.

[14] Trishul M Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project Adam: Building an Efficient and Scalable Deep Learning Training System. In *OSDI*, 2014.

[15] Mosharaf Chowdhury and Ion Stoica. Efficient Coflow Scheduling Without Prior Knowledge. In *SIGCOMM*, 2015.

[16] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient Coflow Scheduling with Varys. In *SIGCOMM*, 2014.

[17] Coflow-Benchmark. https://github.com/coflow/coflow-benchmark, 2020.

[18] Bryce Cronkite-Ratcliff, Aran Bergman, Shay Vargaftik, Madhusudhan Ravi, Nick McKeown, Ittai Abraham, and Isaac Keslassy. Virtualized Congestion Control. In *SIGCOMM*, 2016.

[19] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[20] Mathijs Den Burger, Thilo Kielmann, and Henri E Bal. Balanced Multicasting: High-Throughput Communication for Grid Applications. In *SC*, 2005.

[21] Advait Dixit, Pawan Prakash, Y Charlie Hu, and Ramana Rao Kompella. On the Impact of Packet Spraying in Data Center Networks. In *INFOCOM*, 2013.

[22] Vanini Erico, Pan Rong, Alizadeh Mohammad, Taheri Parvin, and Edsall Tom. Let it Flow: Resilient Asymmetric Load Balancing with Flowlet Switching. In *NSDI*, 2017.

[23] Introducing data center fabric, the next-generation Facebook data center network. https://engineering.fb.com/2014/11/14/production-engineering/introducing-data-center-fabric-the-next-generation-facebook-data-center-network, 2020.

[24] S. Ben Fred, T. Bonald, A. Proutiere, G. Régnié, and J. W. Roberts. Statistical Bandwidth Sharing: A Study of Congestion at Flow Level. In *SIGCOMM*, 2001.

[25] Harold N Gabow and KS Manu. Packing Algorithms for Arborescences (And Spanning Trees) In Capacitated Graphs. *Mathematical Programming*, 82(1):83–109, 1998.

[26] Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, and Mohammad Alizadeh. JUGGLER: A Practical Reordering Resilient Network Stack for Datacenters. In *EuroSys*, 2016.

[27] Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Nikhil Devanur, Janardhan Kulkarni, Gireeja Ranade, Pierre-Alexandre Blanche, Houman Rastegarfar, Madeleine Glick, and Daniel Kilper. ProjecToR: Agile Reconfigurable Data Center Interconnect. In *SIGCOMM*, 2016.

[28] Soudeh Ghorbani, Brighten Godfrey, Yashar Ganjali, and Amin Firoozshahian. Micro Load Balancing in Data Centers with DRILL. In *HotNets*, 2015.

[29] Y. Gong, B. He, and J. Zhong. Network Performance Aware MPI Collective Communication Operations in the Cloud. *IEEE Transactions on Parallel and Distributed Systems*, 26(11):3079–3089, 2015.

[30] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In *NSDI*, 2019.

[31] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. In *SIGCOMM*, 2015.

[32] Apache Hadoop. https://hadoop.apache.org/, 2020.

[33] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. Re-Architecting Datacenter Networks and Stacks for Low Latency and High Performance. In *SIGCOMM*, 2017.

[34] Vipul Harsh, Sangeetha Abdu Jyothi, and P. Brighten Godfrey. Spineless Data Centers. In *HotNets*, 2020.

[35] Sayed Hadi Hashemi, Sangeetha Abdu Jyothi, and Roy Campbell. TicTac: Accelerating Distributed Deep Learning with Communication Scheduling. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *MLSys*, 2019.

[36] Keqiang He, Eric Rozner, Kanak Agarwal, Wes Felter, John Carter, and Aditya Akella. Presto: Edge-based Load Balancing for Fast Datacenter Networks. In *SIGCOMM*, 2015.

[37] Keqiang He, Eric Rozner, Kanak Agarwal, Yu (Jason) Gu, Wes Felter, John Carter, and Aditya Akella. AC/DC TCP: Virtual Congestion Control Enforcement for Datacenter Networks. In *SIGCOMM*, 2016.

[38] Hydro. https://github.com/hydro-project, 2020.

[39] Virajith Jalaparti, Peter Bodik, Ishai Menache, Sriram Rao, Konstantin Makarychev, and Matthew Caesar. Network-Aware Scheduling for Data-Parallel Jobs: Plan When You Can. In *SIGCOMM*, 2015.

[40] Anand Jayarajan, Jinliang Wei, Garth Gibson, Alexandra Fedorova, and Gennady Pekhimenko. Priority-based Parameter Propagation for Distributed DNN Training. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *MLSys*, 2019.

[41] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *ATC*, 2019.

[42] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *NSDI*, 2014.

[43] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Albert Greenberg, and Changhoon Kim. EyeQ: Practical Network Performance Isolation at the Edge. In *NSDI*, 2013.

[44] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be General and Fast. In *NSDI*, 2019.

[45] Nicholas T Karonis, Bronis R De Supinski, Ian Foster, William Gropp, Ewing Lusk, and John Bresnahan. Exploiting Hierarchy in Parallel Computer Networks to Optimize Collective Operation Performance. In *IPDPS*, 2000.

[46] Naga Katta, Aditi Ghag, Mukesh Hira, Isaac Keslassy, Aran Bergman, Changhoon Kim, and Jennifer Rexford. Clove: Congestion-Aware Load Balancing at the Virtual Edge. In *CoNEXT*, 2017.

[47] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. HULA: Scalable Load Balancing Using Programmable Data Planes. In *SOSR*, 2016.

[48] Praveen Kumar, Nandita Dukkipati, Nathan Lewis, Yi Cui, Yaogong Wang, Chonggang Li, Valas Valancius, Jake Adriaens, Steve Gribble, Nate Foster, and Amin Vahdat. PicNIC: Predictable Virtualized NIC. In *SIGCOMM*, 2019.

[49] Katrina LaCurts, Shuo Deng, Ameesh Goyal, and Hari Balakrishnan. Choreo: Network-Aware Task Placement for Cloud Applications. In *IMC*, 2013.

[50] Vinh The Lam, Sivasankar Radhakrishnan, Rong Pan, Amin Vahdat, and George Varghese. Netshare and Stochastic Netshare: Predictable Bandwidth Allocation for Data Centers. *SIGCOMM Comput. Commun. Rev.*, 2012.

[51] Jeongkeun Lee, Yoshio Turner, Myungjin Lee, Lucian Popa, Sujata Banerjee, Joon-Myung Kang, and Puneet Sharma. Application-Driven Bandwidth Guarantees in Datacenters. In *SIGCOMM*, 2014.

[52] He Liu, Feng Lu, Alex Forencich, Rishi Kapoor, Malveeka Tewari, Geoffrey M. Voelker, George Papen, Alex C. Snoeren, and George Porter. Circuit Switching Under the Radar with REACToR. In *NSDI*, 2014.

[53] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. F10: A Fault-Tolerant Engineered Network. In *NSDI*, 2013.

[54] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. Nitrosketch: Robust and General Sketch-Based Monitoring in Software Switches. In *SIGCOMM*, 2019.

[55] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *SIGCOMM*, 2016.

[56] Lpsolve. http://web.mit.edu/lpsolve_v5520/doc/index.htm, 2020.

[57] Liang Luo, Peter West, Jacob Nelson, Arvind Krishnamurthy, and Luis Ceze. PLink: Discovering and Exploiting Locality for Accelerated Distributed Training on the Public Cloud. In *MLSys*, 2020.

[58] William M. Mellette, Rob McGuinness, Arjun Roy, Alex Forencich, George Papen, Alex C. Snoeren, and George Porter. RotorNet: A Scalable, Low-Complexity, Optical Datacenter Network. In *SIGCOMM*, 2017.

[59] Michael Mitzenmacher. How Useful Is Old Information? *IEEE Transactions on Parallel and Distributed Systems*, 11(1):6–20, 2000.

[60] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. In *SIGCOMM*, 2018.

[61] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A Distributed Framework for Emerging AI Applications. In *OSDI*, 2018.

[62] Zhixiong Niu, Hong Xu, Peng Cheng, Qiang Su, Yongqiang Xiong, Tao Wang, Dongsu Han, and Keith Winstein. NetKernel: Making Network Stack Part of the Virtualized Infrastructure. In *USENIX ATC*, 2020.

[63] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. Stateless Datacenter Load-balancing with Beamer. In *NSDI*, 2018.

[64] Pitch Patarasuk and Xin Yuan. Bandwidth Efficient All-reduce Operation on Tree Topologies. In *IPDPS*, 2007.

[65] Y Peng, Y Zhu, Y Chen, Y Bao, B Yi, C Lan, C Wu, and C Guo. A Generic Communication Scheduler for Distributed DNN Training Acceleration. In *SOSP*, 2019.

[66] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. The Design and Implementation of Open vSwitch. In *NSDI*, 2015.

[67] Lucian Popa, Praveen Yalagandula, Sujata Banerjee, Jeffrey C. Mogul, Yoshio Turner, and Jose Renato Santos. ElasticSwitch: Practical Work-Conserving Bandwidth Guarantees for Cloud Computing. In *SIGCOMM*, 2013.

[68] Barath Raghavan, Kashi Vishwanath, Sriram Ramabhadran, Kenneth Yocum, and Alex C. Snoeren. Cloud Control with Distributed Rate Limiting. In *SIGCOMM*, 2007.

[69] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving Datacenter Performance and Robustness with Multipath TCP. In *SIGCOMM*, 2011.

[70] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the Social Network's (Datacenter) Network. In *SIGCOMM*, 2015.

[71] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan R. K. Ports, and Peter

Richtárik. Scaling Distributed Machine Learning with In-Network Aggregation. Technical report, KAUST, Feb 2019. http://hdl.handle.net/10754/631179.

[72] S. Savage, T. Anderson, Amit Aggarwal, David Becker, N. Cardwell, A. Collins, Eric Hoffman, John Snell, Amin Vahdat, G. Voelker, and J. Zahorjan. Detour: Informed Internet Routing and Transport. *IEEE Micro*, 19:50–59, 1999.

[73] Brandon Schlinker, Radhika Niranjan Mysore, Sean Smith, Jeffrey C. Mogul, Amin Vahdat, Minlan Yu, Ethan Katz-Bassett, and Michael Rubin. Condor: Better Topologies Through Declarative Design. In *SIGCOMM*, 2015.

[74] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. Approximating Fair Queueing on Reconfigurable Switches. In *NSDI*, 2018.

[75] Haiying Shen, Ankur Sarker, Lei Yu, and Feng Deng. Probabilistic Network-Aware Task Placement for MapReduce Scheduling. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 241–250. IEEE, 2016.

[76] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P. Brighten Godfrey. Jellyfish: Networking Data Centers Randomly. In *NSDI*, 2012.

[77] Shin-Yeh Tsai and Yiying Zhang. LITE Kernel RDMA Support for Datacenter Applications. In *SOSP*, 2017.

[78] Alexandru Uta, Alexandru Custura, Dmitry Duplyakin, Ivo Jimenez, Jan Rellermeyer, Carlos Maltzahn, Robert Ricci, and Alexandru Iosup. Is Big Data Performance Reproducible in Modern Cloud Networks? In *NSDI*, 2020.

[79] Guanhua Wang, Shivaram Venkataraman, Amar Phanishayee, Nikhil Devanur, Jorgen Thelin, and Ion Stoica. Blink: Fast and Generic Collectives for Distributed ML. In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *MLSys*, 2020.

[80] R. Wang, J. A. Wickboldt, R. P. Esteves, L. Shi, B. Jennings, and L. Z. Granville. Using Empirical Estimates of Effective Bandwidth in Network-Aware Placement of Virtual Machines in Datacenters. *IEEE Transactions on Network and Service Management*, 13(2):267–280, 2016.

[81] Rich Wolski, Neil T. Spring, and Jim Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Future Gener. Comput. Syst.*, 15(5–6):757–768, October 1999.

[82] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *OSDI*, 2018.

[83] Zhuolong Yu, Jingfeng Wu, Vladimir Braverman, Ion Stoica, and Xin Jin. Twenty Years After: Hierarchical Core-Stateless Fair Queueing. In *NSDI*, 2021.

[84] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *EuroSys*, 2010.

[85] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM*, 2016.

[86] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P Xing. Poseidon: An Efficient Communication Architecture for Distributed Deep Learning on GPU Clusters. In *ATC*, 2017.

[87] Hong Zhang, Li Chen, Bairen Yi, Kai Chen, Mosharaf Chowdhury, and Yanhui Geng. CODA: Toward Automatically Identifying and Scheduling Coflows in the Dark. In *SIGCOMM*, 2016.

[88] Hong Zhang, Junxue Zhang, Wei Bai, Kai Chen, and Mosharaf Chowdhury. Resilient Datacenter Load Balancing in the Wild. In *SIGCOMM*, 2017.

[89] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. High-Resolution Measurement of Data Center Microbursts. In *IMC*, 2017.

[90] Yangming Zhao, Kai Chen, Wei Bai, Chen Tian, Yanhui Geng, Yiming Zhang, Dan Li, and Sheng Wang. RAPIER: Integrating Routing and Scheduling for Coflow-aware Data Center Networks. In *INFOCOM*, 2015.

[91] Yangming Zhao, Chen Tian, Jingyuan Fan, Tong Guan, and Chunming Qiao. RPC: Joint Online Reducer Placement and Coflow Bandwidth Scheduling for Clusters. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, 2018.

[92] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion Control for Large-Scale RDMA Deployments. In *SIGCOMM*, 2015.

[93] Siyuan Zhuang, Zhuohan Li, Danyang Zhuo, Stephanie Wang, Eric Liang, Robert Nishihara, Philipp Moritz, and Ion Stoica. Hoplite: Efficient and Fault-Tolerant Collective Communication for Task-Based Distributed Systems. In *SIGCOMM*, 2021.

Tiara: A Scalable and Efficient Hardware Acceleration Architecture for Stateful Layer-4 Load Balancing

Chaoliang Zeng[1*] Layong Luo[2] Teng Zhang[2] Zilong Wang[1*] Luyang Li[3*] Wenchen Han[4*]
Nan Chen[2] Lebing Wan[2] Lichao Liu[2] Zhipeng Ding[2] Xiongfei Geng[2] Tao Feng[2]
Feng Ning[2] Kai Chen[1] Chuanxiong Guo[2]

[1]*Hong Kong University of Science and Technology* [2]*ByteDance* [3]*ICT/CAS* [4]*Peking University*

Abstract

Stateful layer-4 load balancers (LB) are deployed at datacenter boundaries to distribute Internet traffic to backend real servers. To steer terabits per second traffic, traditional software LBs scale out with many expensive servers. Recent switch-accelerated LBs scale up efficiently, but fail to offload a massive number of concurrent flows into limited on-chip SRAMs.

This paper presents Tiara, a hardware architecture for stateful layer-4 LBs that aims to support a high traffic rate (> 1 Tbps), a large number of concurrent flows (> 10M), and many new connections per second (> 1M), without any assumption on traffic patterns. The three-tier architecture of Tiara makes the best use of heterogeneous hardware for stateful LBs, including a programmable switch and FPGAs for the fast path and x86 servers for the slow path. The core idea of Tiara is to divide the LB fast path into a memory-intensive task (*real server selection*) and a throughput-intensive task (*packet encap/decap*), and map them into the most suitable hardware, respectively (i.e., map *real server selection* into FPGA with large high-bandwidth memory (HBM) and *packet encap/decap* into a high-throughput programmable switch). We have implemented a fully functional Tiara prototype, and experiments show that Tiara can achieve extremely high performance (1.6 Tbps throughput, 80M concurrent flows, 1.8M new connections per second, and less than 4 us latency in the fast path) in a holistic server equipped with 8 FPGA cards, with high cost, energy, and space efficiency.

1 Introduction

Large service providers deploy various services inside their geo-distributed datacenters of different scales. At the boundary of these datacenters, stateful layer-4 load balancers (LB), a.k.a., multiplexers (Mux), are deployed to distribute user requests from the Internet to many real servers inside datacenters while preserving connection consistency. Driven by

exponentially increased content delivery and cloud computing demands, a typical LB in large service providers usually has to process terabits per second of Internet traffic, with tens of millions of concurrent flows [25, 31] and millions of new connections per second (CPS) [12].

To support such high performance, vendor-proprietary hardware LBs (e.g., F5 [9]) were deployed in the early days of some datacenters. However, they lacked agility, which is highly desirable in modern hyper-scale datacenters. In recent years, the move from vendor-proprietary hardware to in-house software LBs, or software Muxes (SMux), e.g., Ananta [36] and Maglev [21], was mainly driven by requirements like manageability, reliability, and agility, but sacrificed efficiency (i.e., cost, energy, and space efficiency). For example, Ananta [36] achieves 10 Gbps per instance, and supporting up to terabits per second throughput requires scale-out with a large number of servers. Deploying so many servers for just LB is not only costly but also challenging at energy- or space-limited boundaries of massive small/medium-scale datacenters (e.g., 10s-100s of servers in PoPs [15] or edge [40]). Moreover, software LBs usually suffer from high latency and jitter, sometimes comparable to Internet access latency (in the order of milliseconds [24]) when CPU load is high. Such latency churn will adversely impact users' network experience.

To improve the efficiency of software LBs without sacrificing agility, there is an emerging trend to accelerate software LBs with in-house software and hardware co-design. Recent work [16, 23, 24, 31] leverages programmable switches to accelerate LBs. Nevertheless, programmable switches have inherent scalability issues (§2.3). *On the data plane*, a modern switch cannot store a large number of concurrent flows due to its small memory size (typically 50-100 MB on-chip SRAMs); *on the control plane*, the switch fails to support a large CPS given its slow entry insertion speed (∼ 100 Kps).

Existing switch-accelerated LBs do not address both challenges simultaneously. For example, Silkroad [31] stores a small hash of a connection instead of the 5-tuple to compress the connection table. However, its scalability is still bounded by the switch's small memory size, and it may suffer from

---

throughput reduction due to switch pipeline folding. Moreover, Silkroad does not address the scalability problem on the control plane. Cheetah [16] provides a fast entry insertion mechanism by storing an index in the packet header but requires modifications on services' client sides. Thus, applying such a mechanism is difficult, if not impossible, in the datacenter with thousands of services [19, 36]. Furthermore, it does not address the scalability issue on the data plane.

One plausible approach to address the above problems is to leverage traffic locality in hardware offloading. If the traffic pattern follows a long-tail distribution (i.e., a small number of flows carry the majority of the traffic), only a few elephant flows need to be offloaded and stored in the switch, thus lowering the requirements of both the hardware memory size and entry insertion speed. However, we observe from production datacenters that the traffic patterns at datacenter boundaries do not necessarily follow a long-tail distribution. Instead, the mix of VIP traffic for multiple services is highly dynamic and unpredictable, detailed in §2.2.

Based on the above analysis and observation, we ask: *can we design a scalable and efficient stateful LB without any assumption on traffic patterns?* Specifically, the design should be:

- **scalable** on both data plane (store > 10M concurrent flows) and control plane (support > 1M CPS);

- **efficient** in terms of high cost, energy, and space efficiency; and

- **generic** without any assumption on traffic patterns.

To this end, we move one step further beyond the existing *switch-server* architecture [23, 24] by exploring more flexible hardware, i.e., FPGA. FPGA is a high-performance and programmable device becoming an important building block in the datacenter infrastructure [22, 28, 29, 42]. The modern FPGA equipped with gigabytes of high-bandwidth memory (HBM) is well-suited to improve LB scalability, as HBM can store a large number of concurrent flows with high lookup and insertion rate.

In this paper, we present Tiara, a three-tier hardware acceleration architecture composed of a programmable switch, FPGAs, and commodity servers, for a high-performance stateful LB with scalability and efficiency. The core idea behind Tiara is that we map different LB tasks into different devices by matching task requirements with device capabilities (§3.1). Specifically, Tiara divides the LB fast path into *real server selection*, a memory-intensive task with both large capacity and high bandwidth requirements, and *packet encap/decap*, a throughput-intensive task. Then, Tiara maps these two tasks into FPGAs with large HBM and a programmable switch with high packet processing throughput, respectively. Similar to other hardware-accelerated systems [23, 24, 37], Tiara leverages commodity servers as the slow path to handle the unprocessed traffic from the fast path.

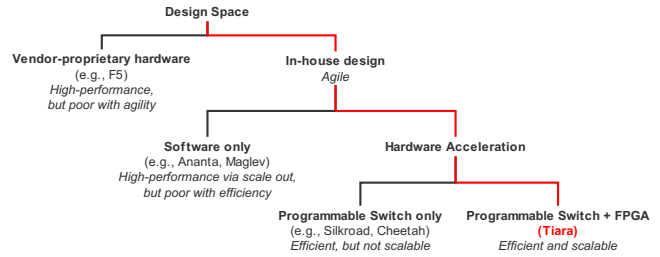To support high CPS without compromising line-rate

Figure 1: Design space for stateful LB architectures.

packet processing in a heterogeneous system, we optimize several key design components in Tiara (§3.3). First, for both fast lookup and insertion, Tiara adopts fixed-length hash chaining, which leverages the parallel processing capability in both FPGAs and multi-core servers. Second, we design a lock-free offloading approach to support issuing millions of entry operations per second from a server to an FPGA. Third, Tiara employs a lightweight aging mechanism to recycle outdated entries, where FPGAs periodically report connection activeness via a dedicated accessing bitmap, preventing interference with the data plane.

We have implemented a fully functional Tiara prototype based on a Barefoot Tofino switch, a Xilinx FPGA-based SmartNIC card, and a commodity server. We modified a production-level SMux for the slow path and the control plane (§4). The key results from our experiments (§5) show that our prototype can support 10M concurrent flows and 1.8M CPS, 9× better than Silkroad [31], at 200 Gbps with less than 4 us average latency and small jitter in the fast path. In a holistic server with 8 FPGA cards, Tiara can provide superiority in throughput (up to 1.6 Tbps) and flow capacity (up to 80M concurrent flows). Meanwhile, Tiara achieves 17.4×, 12.8×, and 16.8× higher cost, energy, and space efficiency, respectively, compared to SMux.

As a summary, Figure 1 shows the design space for stateful LB architectures and the unique position of Tiara. Tiara is more agile than traditional vendor-proprietary hardware, faster and more cost-, energy-, and space-efficient than software LBs, and more scalable than switch-accelerated solutions. Specifically, Tiara makes the following contributions:

- We propose a three-tier architecture that matches key LB tasks to the most suitable hardware: programmable switch for packet encap/decap, FPGA with HBM for connection management, and x86 CPU for SMux.

- We design and optimize key LB components, including an efficient hash table structure for fast lookup and efficient insertion, a lock-free offloading approach to improve connection offloading speed, and a lightweight aging mechanism with little overhead and minimal interference on the data plane.

- We implement the Tiara prototype and conduct testbed experiments to show its performance superiority.

2 Background

2.1 Layer-4 Load Balancing

Layer-4 LB can be classified into the stateful LB, which stores the connection-to-real server (RS) mapping as a connection table (CT), and the stateless LB, which does not maintain any per-connection state. Most of the industry LBs are stateful [3, 5, 8, 21, 36] because stateful LBs can easily ensure *per connection consistency (PCC)* [16, 31], which means all packets of a connection should be delivered to the same RS to avoid breaking the connection. In this paper, we focus on the stateful LB, which usually contains the following two parts.

Real server selection: The LB selects an RS for each incoming packet by identifying its connection via the 5-tuple in the packet header. The LB selects RS in two ways. For the first packet of a connection, the LB selects an RS based on a pre-defined algorithm, e.g., hash, round-robin, or least-loaded, and creates a connection entry in the CT to record this selection. The LB selects the same RS for the other packets of this connection by looking up the CT. This mechanism ensures PCC. An RS can be specified by a tuple of {RS_IP, RS_Port} based on backend service implementations.

Packet encap/decap: After an RS is selected for an inbound packet, the LB encapsulates the packet with RS_IP and RS_Port. The encapsulation process may include multiple steps in practice. Given a tuple of {RS_IP, RS_Port}, the LB enforces Port NAT (virtual Port (VPort) → RS_Port), IP NAT (virtual IP (VIP) → RS_IP), and packet encapsulation with VxLAN. Unlike inbound traffic processing involving both RS selection and packet encapsulation, outbound traffic processing only needs packet decapsulation.

2.2 Nature of Internet traffic at the Datacenter Boundary

Large service providers usually deploy many Internet services in a datacenter, and the Internet traffic at the datacenter boundary is a mix of multiple services' traffic, with the following properties.

The flow distribution of individual services varies. The distribution of service traffic depends heavily on the service's client- and server-side implementations. For example, certain service clients may split an elephant flow into multiple smaller ones to reduce the cost of TCP disconnection over unstable Internet, leading to a uniform distribution. In contrast, other service clients may use short connections for synchronization and long connections for massive data transmission, leading to a long-tail distribution. To show this fact, we analyze flow distributions for three different services, as shown in Figure 2. These three services have various flow distributions: service C shows a uniform distribution (where top 10% flows carry 19.6% traffic), while service A and B exhibit traffic locality
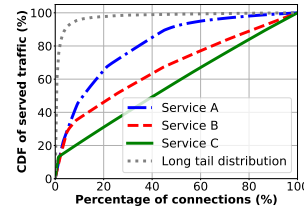

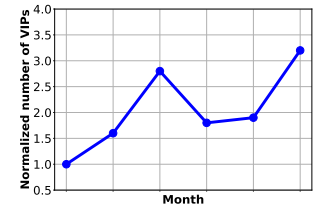Figure 2: The traffic distribution varies among three different services.

Figure 3: The number of VIPs in a typical LB. It shows a high variation over 6 months.

(where top 10% flows carry 46.3% and 35.5% traffic, respectively) to different extents.

The traffic volume of a service can dynamically change. The traffic volume of an individual service keeps changing independently, with different short-term daily peaks and troughs [21] and long-term uncertainty due to the change in user interest [20]. At any given time in the mixed service traffic, mice flows of one service at peak might consume more bandwidth than elephant flows of another service at the trough, making the overall distribution of their mix unpredictable.

The number of VIPs at a datacenter boundary can change over time. Large service providers keep launching, stopping, and migrating services, driven by various reasons like changes in user interest or business opportunities. Figure 3 reveals a high variation (3.2×) of the number of VIPs served by an LB over 6 months. The dynamic change of services inside the datacenter further makes the mixed VIP traffic at the boundary highly dynamic without any specific distribution.

Based on these observations, we should not rely on any assumption of specific traffic distributions (e.g., long-tail distribution) when designing load balancers at datacenter boundaries for mixed services.

2.3 Accelerating LB with Programmable Switches

Most recent proposals accelerate LBs by realizing hardware Muxes (HMux) [23, 24, 31] with programmable switches, where the RS selection and packet encapsulation are implemented in switch processing pipelines. HMuxes can effectively reduce the number of required servers, which is significant, especially for small/medium-scale datacenters. Nevertheless, using programmable switches as HMuxes suffers from scalability issues on both data and control planes.

Data plane: As widely discussed, switching ASICs cannot support many concurrent flows due to their limited memory sizes [24,25,31,37]. Considering a CT with an entry size of 64 bytes[1] and a typical concurrent flow number of 10M [25, 31],

---

[1]64 bytes/entry is an empirical value for IPv6, including 37 bytes for the
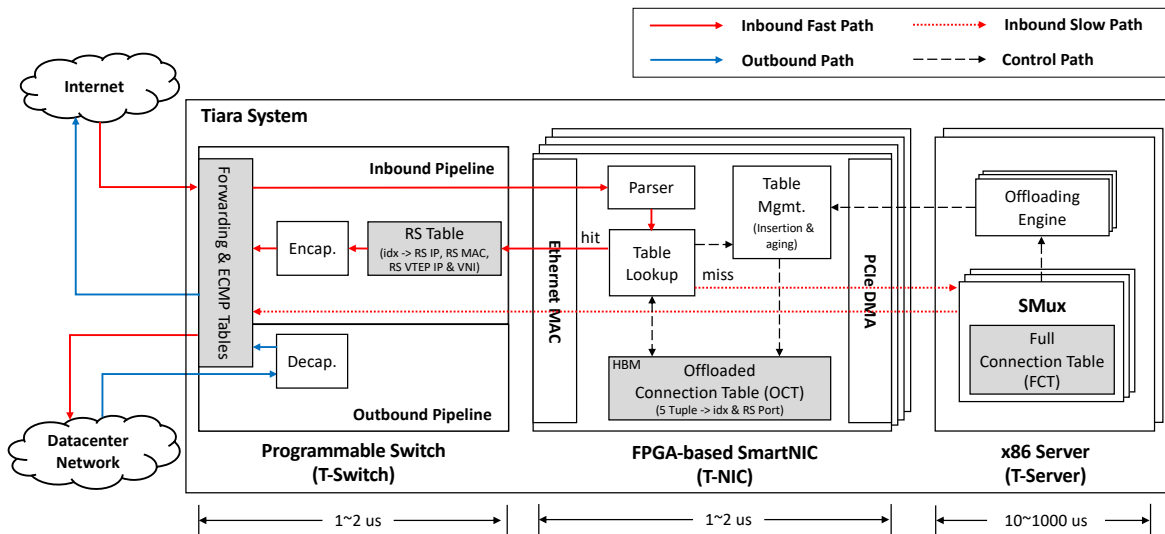
Figure 4: Tiara architecture. Tiara consists of three tiers: T-switch, T-NIC, and T-server. Tiara divides LB into multiple key tasks and matches them respectively to suitable hardware tiers: T-switch for stateless packet encap/decap, T-NIC with HBM for connection lookup and management, and T-server as a last resort.

the CT size is 640 MB. However, modern programmable switches only provide 50-100 MB SRAMs [31]. Moreover, these SRAMs are typically distributed into multiple pipelines, e.g., 15 MB/pipeline. To look up a larger table than a single pipeline's SRAM size, HMuxes typically use folded pipelines and resubmit a packet to switch pipelines via different physical ports, reducing the available throughput.

Control plane: State-of-the-art programmable switches are slow for entry insertion. For example, a Barefoot Tofino switch can only do $\sim$ 100K insertions per second after our optimizations. We measure the entry insertion overhead. Our result reveals that the top two time-consuming functions are the hash computation and the Cuckoo search algorithm [34]. Our result is similar to those of previous work [16, 31]. The root causes exist in the low-end switch CPU, slow PCIe interconnect between the CPU and the switching ASIC, and the small memory size in the switching ASIC. The first two factors affect the speed of hash computation and operation offloading. Then the limited memory space forces the switching ASIC to rely on space-efficient Cuckoo hashing for hash collision resolution. The Cuckoo hashing impedes fast insertion by (1) multiple entry movements during collision resolution and (2) incapability of parallelization due to the dependency between two insertions (the previous insertion location may affect the latter one). The above hardware constraints make it difficult for a switch to support > 1M CPS required by production LBs [5].

3 Tiara Design

We now present Tiara, a novel hardware-accelerated LB architecture, which can support > 1 Tbps traffic, > 10M concurrent flows, and > 1M CPS, without any assumption on traffic patterns.

3.1 Architecture Overview

Tiara is a three-tier architecture as demonstrated in Figure 4. The outermost tier is a programmable switch (T-switch), which sits between the Internet and the datacenter network as a bump in the wire. The second tier is a group of FPGA-based SmartNICs (T-NIC), which act as the HMux jointly with T-switch for LB fast path. The third tier consists of commodity servers (T-server), which host T-NICs and implement SMuxes for LB slow path. The number of T-NICs hosted by a T-server and the number of T-servers behind T-switch are configurable, making the three-tier architecture flexible enough to meet different performance requirements at various datacenter entrances.

The novel idea of Tiara is that it maps different LB tasks into their most suitable devices based on their unique capabilities. In the fast path, Tiara divides the HMux between T-NICs and T-switch. Tiara leverages the large and fast HBM inside T-NIC's FPGA for memory-intensive *RS selection*. One typical HBM stack comprises 16 256-MB memory channels, and each channel provides $\sim$100 million lookups per second (MLPS)[2]. To maximize the accessing performance, we should separate memory accesses to different memory channels. The

---

5-tuple as match key, 18 bytes for `RS_IP` and `RS_Port` as action data, and a few bytes for packing and alignment overhead.

[2]One memory channel provides $\sim$100 million random read accesses per second based on our emulation [1].

parallelism among HBM channels is carefully explored to meet memory capacity and throughput requirements of RS selection, which will be discussed in §3.3.1. Meanwhile, Tiara leverages the high performance and programmability properties of T-switch pipelines for throughput-intensive *packet encap/decap*.

Besides the fast path processing, Tiara instantiates several SMuxes in T-server to act as a backstop for unprocessed traffic. Each SMux maintains a full connection table (FCT) for all inbound flows, and is associated with a T-NIC virtual function with dedicated DMA channels used for packet receiving and sending.

Programmable switch or fixed-function switch. Another option of Tiara's three-tier architecture demonstrated in Figure 4 is that the programmable T-switch could be replaced by a fixed-function switch that only performs forwarding and ECMP routing. If so, the switch packet processing logic, including RS table, packet encapsulation, and decapsulation, can be moved into T-NICs. We do not choose this option for a few reasons. First, the performance, cost, and power consumption of programmable switches is comparable to that of traditional fixed-function switches [14]. Second, with packet decapsulation implemented in programmable T-switch, the architecture allows outbound traffic to bypass T-NICs (as described in §3.2.2), thus halving the T-NICs bandwidth requirements and the number of required T-NICs. Third, the programmability of T-switch relieves T-NIC implementation. If all fast path functions are implemented in T-NIC, it will increase not only the FPGA size, power consumption, and cost, but also the development time, as programming switches with P4 is easier than programming FPGA with Verilog.

3.2 Control & Data Planes

3.2.1 Control Plane

A typical LB usually includes a centralized controller configuring $VIP \rightarrow RS\_IP$ mappings into Muxes and BGP speakers for VIP announcements. As they are common and well described in previous work [21, 23, 24, 36], we will skip them in this paper and pay more attention to the acceleration-related control flow, i.e., the connection management between software and hardware. Tiara relies on T-servers to make the local control plane decisions, including the CT entry insertion and the entry recycling (connection aging). The powerful CPU prevents inefficient hash computations like that on the switch-based HMux. T-servers use *offloading engines* to offload the entry operations generated by SMuxes, to a specific T-NIC, and each offloading engine is associated with a dedicated DMA channel for entry operations. To efficiently process entry insertion and aging, a few optimizations are made in offloading engines, which will be discussed in §3.3.

Moreover, Tiara integrates many more features like management, telemetry, and fault tolerance in the control plane.

Except for the telemetry, Tiara can support all these functionalities solely in the control plane. Network telemetry requires collecting statistic counters from the data plane, and T-NIC and T-switch can provide them easily without affecting the fast path performance.

3.2.2 Data Plane

Inbound fast path. Upon receiving a packet from the Internet, T-switch distributes it to one of the T-NICs based on ECMP. Then, T-NIC parses the packet header and uses the extracted fields (i.e., 5-tuple) to look up the offloaded connection table (OCT), which maintains up to tens of millions of connections in FPGA HBM and sustains fast lookup. The lookup result from OCT is an LB decision, i.e., a two-tuple {RS_Index, RS_Port}, where RS_Index represents a real server and will be used in later RS table lookup in T-switch. Instead of replacing the RS_Port locally, which will incur checksum computation, Tiara delays this operation to T-switch processing. T-NIC encapsulates the retrieved tuple into a packet metadata header between the Ethernet header and the IP header, and sends back the packet to T-switch. T-switch looks up an RS table and gets the corresponding RS information, including RS_VTEP_IP, RS_MAC, RS_IP, and VNI. Finally, T-switch enforces Port NAT, IP NAT, and VxLAN encapsulation sequentially, and forwards the encapsulated packet to the RS. Since we decouple the RS_Port from the RS table, the number of entries in the RS table is the same as the number of real servers, typically 10K-100K[3]. Compared to CT, the RS table is relatively stable, updated in second time granularity [36], which is far slower than the entry insertion speed provided by T-switch. Based on these two features, the RS table is achievable in the T-switch SRAMs.

Inbound slow path. When a packet misses in the OCT, the T-NIC uploads it to an SMux via a PCIe DMA channel chosen by *Receive Side Scaling* (RSS). Upon receiving the packet, the SMux looks up the FCT and moves to one of the following two workflows according to the lookup result.

If the packet belongs to an established connection, it will hit in the FCT lookup. SMux retrieves the corresponding {RS_Index, RS_Port}, and further processes the encapsulation for this packet by looking up the RS table locally[4]. Finally, SMux sends the encapsulated packet to the real server (via T-NIC and T-switch). There is a trick on VxLAN source port calculation. Since the source port is calculated by hashing [13], SMux reuses the last 2 bytes of RSS hash value from the T-NIC to avoid duplicate hash computation.

If it is the first packet of a new connection, it will miss in the FCT lookup. SMux makes the LB decision to create a connection entry for this connection and inserts the generated

---

[3]A typical datacenter supports thousands of services [19, 36], and each one usually holds 10-100 instances.

[4]In fact, these two tables can fuse into one table.

entry into the FCT. Then, SMux encapsulates the packet and sends it out.

In both cases, SMux will try to insert the corresponding connection entry into OCT. If there are empty slots in the corresponding hash bucket in OCT, the insertion will be successful; otherwise, Tiara will fail the insertion without cache eviction and keep that flow in SMux. We leave the cache eviction policy for the LB connection table as future work.

Outbound path. For outgoing packets, real servers leverage XDP [4] or OVS *Conntrack* [10] to perform SNAT locally. The real servers rewrite source IP with `VIP` and source ports with `VPort`, and forward the packets in VxLAN encapsulation to T-switch. T-switch further performs packet decapsulation and sends the packets to the Internet. As the only LB operation (i.e., packet decapsulation) for outbound packets is offloaded completely in T-switch, outbound traffic can bypass T-NICs and SMuxes, halving the T-NICs bandwidth requirements.

3.3 Component Design & Optimization

3.3.1 Efficient Hash Table Structure

The hash table design of OCT affects not only lookup performance in hardware but also entry insertion speed in software. We leverage an efficient hash table structure that enables both fast lookup in T-NIC and fast entry insertion in T-server. Specifically, we expect the hash table used in T-NIC should (1) support O(1) and parallel insertions in software and (2) support line-rate lookup in hardware.

We observe that a hash table with fixed-length chaining can satisfy all requirements. First, the insertion complexity of hash chaining is O(1). Second, since the hash computations of different insertion indexes are independent, we can utilize multiple cores in T-server to compute the insertion indexes in a parallel manner. Third, T-NIC can support O(1) lookup by mapping fix-length chains into multiple HBM channels. Last, fix-length hash chaining simplifies hardware design. If using variable-length hash chaining, dynamic memory management is mandatory and unfriendly to hardware implementation.

T-NIC manages OCT using a hash table with fixed-length chaining, as illustrated in Figure 5. Despite the simple structure, determining the proper parameters of the hash table in HBM to achieve both fast lookup and low collision rate is non-trivial. For the hash table with fixed-length chaining, two parameters control the shape of the table: the number of hash indexes (*depth*) and the number of entries at each index (*width*), following that $depth \times width = hash\ table\ size$. Given a fixed hash table size, a deeper hash table results in a higher hash collision rate (see analysis in Appendix A), while a wider hash table poses challenges for line-rate lookup on HBM, as the number of parallel HBM memory channels is limited.

Based on the above analysis, T-NIC determines the hash table parameters with a principle that *maximizing the width*
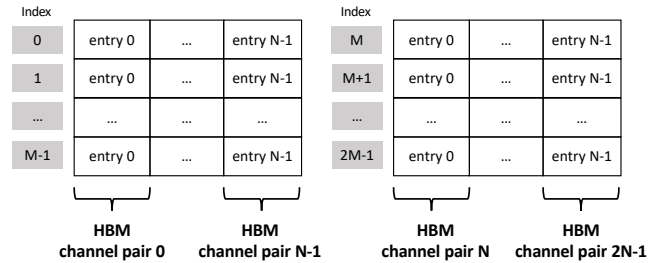

Figure 5: The fixed-length hash chaining design in Tiara OCT, where depth is *M* and width is *N*. Each channel pair saves a column of the hash table. For a table lookup, T-NIC can launch multiple parallel accesses of *N* entries inside 2*N* HBM channels.

while guaranteeing line-rate lookup. Take the FPGA card used in our implementation (§4) as an example. It has two 100GE ports, each requiring 150 MLPS to sustain line rate, and one HBM stack of 16 256-MB (8M × 256-bit width) memory channels, each providing up to 100 MLPS. We divide 16 channels evenly between two ports so that there are 8 channels to support 100 Gbps traffic. Moreover, the entry size is 512 bits, as discussed in §2.3, so we need to pair two channels for one entry access and construct 4 channel pairs for each port. Given that each channel pair can support 8M entries, there are three candidate hash table structures: 8M (depth) ×4 (width), 16M×2, 32M×1, where one lookup operation involves 4, 2, and 1 channel pair(s), respectively. However, the lookup performance of the 8M×4 hash table structure is only 100 MLPS (using all channels for one lookup), failing to support the 100 Gbps line rate. Based on the principle, the best hash table structure for one 100GE port is 16M×2 in our FPGA card.

T-NIC relies on the connected T-server to simplify hash collision resolution. When there is a hash collision in the table lookup, T-NIC will forward the packet to the slow path in T-server; when there is a hash collision in the entry insertion, the insertion fails in the offloading engine (§3.3.2), and that flow will be kept in the slow path. As long as the hash collision rate is low (2.6% in theory for 10M flows in the 16M×2 hash table), hash collision does not have significant performance penalty.

3.3.2 Lock-free Offloading Approach

We design a lock-free offloading approach to enable issuing millions of insertion or deletion operations per second from SMuxes to T-NIC, which is required to support > 1M CPS.

In Tiara, SMuxes offload the generated entry operations, including entry insertion and deletion, to T-NICs via offloading engines. Given the multi-channel feature of our PCIe DMA engine, Tiara instantiates a few offloading engines and associates each with a dedicated DMA channel, so that offloading engines can offload entries independently.
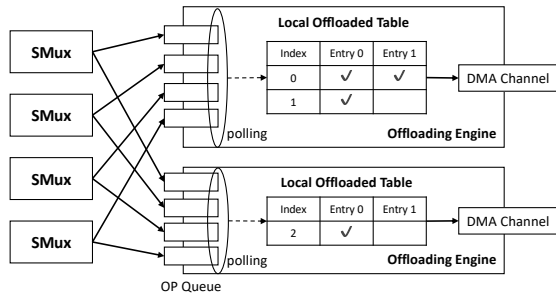
Figure 6: Tiara's lock-free offloading design.

A straightforward offloading approach introduces locks in two places. The first lock happens when multiple SMuxes are mapped to the same offloading engine with only one OP queue. SMuxes can write their operations to the OP queue only when they retrieve a write lock. The second lock exists when multiple offloading engines insert entries into the same hash index. A lock is required for unavoidable synchronizations on a global OCT, maintained in the server to track the OCT usage among different offloading engines. These two locks prevent us from fully leveraging the parallelism in both the multi-core server and the multi-channel DMA to achieve fast entry offloading.

We design a lock-free offloading mechanism, as shown in Figure 6. First, to realize lock-free entry delivery from SMux to the offloading engine, Tiara sets up an OP queue for each SMux-engine pair. The offloading engine polls OP queues in a round-robin manner to retrieve the offloading operations. Second, Tiara adopts the mapping method based on the entry's hash index, i.e., *index-to-engine* mapping. Entries inserted into the same place are delivered to the same offloading engine. Consequently, different offloading engines handle entries with different hash indexes, and each offloading engine maintains a local OCT to track the offloaded indexes. Since the local OCTs are disjoint with each other, it is lock-free during the table update.

For each entry operation, offloading engines will notify SMuxes whether the operation is successful or not (an insertion will fail when the corresponding hash bucket is full) via completion queues (not shown in Figure 6).

3.3.3 Lightweight Aging Mechanism

The purpose of this component is to recycle outdated entries in the OCT, i.e., when a connection is disconnected, its related entry in the OCT should be released so that it can be reused for new connections. To realize it, we need to detect the close of connections. One naive method is to use the TCP FIN packet as the signal of the connection close, which can be captured in T-NICs. However, this method fails on abnormal close of TCP traffic and connection-free UDP traffic.

To unify the flow removing process for TCP and UDP, Tiara adopts an entry aging mechanism that removes a flow entry

from the OCT if it is not accessed in a period T. This aging mechanism may kick out connections whose access interval is larger than T by mistake, but those connections can be further processed in the slow path FCT[5].

The challenge of this aging mechanism is to monitor the accessing states of 10M connection entries periodically with a small memory footprint, minimal performance interference on the data plane, and low CPU overhead.

To address this challenge, T-NIC leverages an accessing bitmap to track connection activities, signals activities to SMuxes, and SMuxes make aging decisions by issuing entry deletion operations based on signals.

T-NIC maintains the accessing bitmap in on-chip SRAMs, using each bit as an indicator for a connection entry. All indicators are reset to 0 at the beginning of every detection period Δ_t $(< T)$. An indicator will be marked as active, i.e., set to 1, only if a packet is accessing the corresponding connection entry. As an active signal, the packet header will be sent to an SMux by RSS, ensuring that the same SMux processes both teardown and establishment for a connection. Subsequent packets accessing active connection entries neither change the indicator status nor trigger signaling to SMuxes. In this way, if the connection is active in a detection period, the related SMux will get a signal. If the SMux does not receive any signal for a connection in multiple continuous (T/Δ_t) periods, that connection is considered outdated and should be aged. T-NIC leverages the length of the detection period to control the reporting frequency, which balances the SMux load and the detection precision.

This mechanism is lightweight in three aspects. First, the memory footprint used for tracking connection states in FPGA is minimal, with one bit per connection in the bitmap. Second, as the accessing bitmap is stored in on-chip SRAMs, the aging process will not interfere with HBM lookup in the fast path. Third, given the low signaling frequency (likely to be minutes level), the PCIe and CPU overhead are both low.

4 Implementation

We implement a fully functional prototype of Tiara with one T-switch and one T-server, equipped with one T-NIC through a PCIe Gen3 x16 link. T-switch and T-NIC are connected via 100G Ethernet cables. In the rest of this section, we discuss the implementation details of each component.

4.1 T-switch

We build a P4 prototype of T-switch with a Barefoot Tofino switch, where one pipeline has 12 physical stages, each with 1.25 MB SRAMs and 528 KB TCAMs.

---

[5]The aging procedure in the FCT is implemented by the SMux, which should provide a longer life cycle for a typical entry compared to the OCT due to its larger memory space.
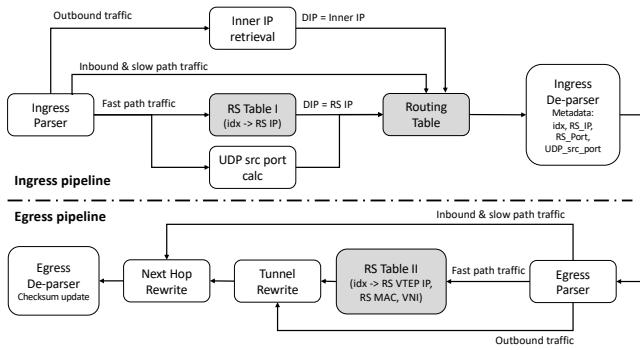
Figure 7: T-switch pipeline implementation.

Figure 8: The metadata format.

We modify a baseline switch.p4[6] to implement the packet processing pipeline, including the RS table, routing tables (forwarding table and ECMP table), and tunnel processing (VxLAN encapsulation and decapsulation). Figure 7 shows an overall pipeline of T-switch. It is worth mentioning that we split the RS table into two parts. The RS table I (`RS_Index` → `RS_IP`) exists in the ingress pipeline, where T-switch retrieves `RS_IP` for routing tables lookup. T-switch postpones the lookup of the rest values in RS table II (`RS_Index` → `RS_VTEP_IP`, `VNI`, `RS_MAC`), to the egress pipeline. This decoupling helps mitigate resource contention between RS Table and routing tables in the ingress pipeline.

The modified switch.p4 takes 53.85% of SRAMs and 13.19% of TCAMs to implement the pipeline described in Figure 7 with 64K RS table entries, 2K IPv4 addresses, 1K IPv4 prefixes, 1K IPv6 addresses, and 1K IPv6 prefixes.

Recall that, in slow path processing, the VxLAN source port is computed based on the last 2 bytes of RSS value (§3.2.2). To be consistent with the slow path, the fast path in T-switch should compute this field in the same way. However, T-switch pipeline does not support the Toeplitz hash [18, 26, 30] used in RSS computation[7]. To address this issue, T-NIC carries the computed RSS value (last 2 bytes) on packets within an extended metadata header to T-switch (§4.2). T-switch retrieves the hash value from the packet and performs the same computations as SMuxes. In our implementation, the VxLAN source port is computed as followed: $port = (RSS \wedge (65535 - 49152)) + 49152$.

---

[6]A simplified version can be found at https://github.com/p4lang/switch

[7]We follow the standard RSS computation procedure for compatibility

4.2 T-NIC

T-NIC is implemented in a Xilinx FPGA-based SmartNIC card, with two 100GE ports and one HBM stack of 16 256MB memory channels. We use Xilinx QDMA IP [11] as the DMA engine. We implement the T-NIC logic described in Figure 4, in System Verilog, including the OCT management and lookup, packet metadata encapsulation, entry aging, and the slow path delivery.

Tiara relies on a metadata header in the packet to pass information between T-NIC and T-switch. Figure 8 shows the format of the metadata header, which is inserted between the Ethernet header and the IP header. The metadata header includes a 4-byte `RS_Index`, a 2-byte `RS_Port`, a 2-byte `RSS`, and a 2-byte `EtherType`. The field `EtherType` in the metadata header follows the IEEE 802 standard [6] to indicate the following header type (IPv4 or IPv6). In the Ethernet header, the original `EtherType` field is changed to 0x88B5, which indicates the next header is private. To avoid the drop of oversized packets caused by inserting the metadata header, we increase the MTU of T-NIC and the corresponding T-switch ports by 10 bytes, i.e., the size of the metadata header.

4.3 T-server

T-server contains 2 Intel(R) Xeon(R) Platinum 8260 CPU. We run SMuxes and offloading engines in one CPU in the same NUMA node as T-NIC without hyper-threading. We build a T-NIC driver as a DPDK [2] PMD and implement the offloading engine on top of it. We leverage an in-house SMux implementation modified from DPVS [3]. The SMux has been deployed over three years, and we make necessary changes to adapt it for the Tiara architecture. The hash computation used in both SMuxes and T-NICs is the CRC32 algorithm.

We optimize the DMA transmission between T-NIC and the PMD. QDMA is a type of Scatter-Gather DMA from Xilinx [11]. For any DMA transaction, it first reads a descriptor from the host to get the physical address of the DMA buffer. The speed of descriptor filling affects the DMA performance. Tiara leverages SIMD instructions provided by Intel processors [7] to accelerate the descriptor filling. For example, we use `_mm_storeu_si128` and `_mm_storeu_si128` to copy the DMA information between the DPDK `mbuf` and the QDMA descriptor. Moreover, Tiara decouples DMA control channels from data channels to avoid head of line blocking and mutual interference. Tiara guarantees lossless control channels by fine-grained credit control between T-server and T-NIC while remaining data channels to be lossy like conventional NIC data paths.

5 Evaluation

In this section, we use testbed experiments to evaluate the Tiara prototype as described in §4. We first show the micro-
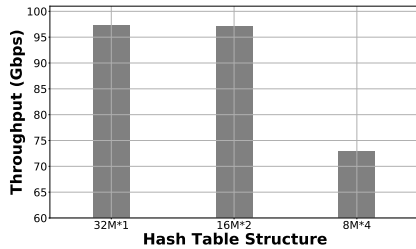
Figure 9: HBM lookup performance on different hash table structures with 10M flows.
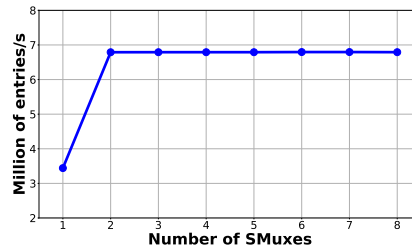

Figure 10: Entry insertion speed of a single offloading engine.
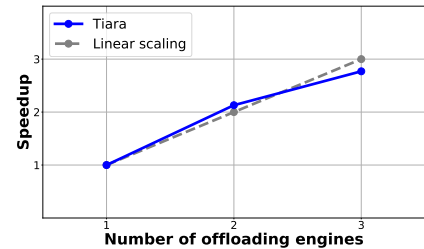

Figure 11: Insertion speedup with multiple offloading engines.

benchmarks to assess the effectiveness of Tiara component designs (§5.1). Then, we measure the end-to-end system performance of Tiara (§5.2). Last, we compare Tiara with existing approaches, i.e., SMux and Silkroad [31] (§5.3). Our results reveal that:

- A T-server with a single T-NIC can provide 200 Gbps throughput with 10M concurrent flows and could scale linearly up to 1.6 Tbps and 80M concurrent flows by hosting 8 T-NICs within a T-server.

- Tiara fast path can provide less than 4 us average latency with small jitter even at line rate.

- Tiara can serve up to 1.8M new connections per second, which is larger than switch-based HMux.

- Tiara is cost-, energy-, and space-efficient, costing $17.4\times$ less money, consuming $12.8\times$ less energy, and taking $16.8\times$ less rack space than SMux, given the same target throughput.

Testbed setup. We leverage the same SMux used in the Tiara slow path as the baseline of software LBs. The Tiara prototype and the baseline are directly connected to the traffic generator using 100 Gbps cables. Test traffic is generated by a hardware traffic generator, sent to the LB (Tiara or baseline), and then routed back to the generator. In this way, we could test the throughput and latency for both Tiara and the baseline.

Traffic. We use a hardware generator to inject synthetic TCP/UDP flows. Since we do not hold any assumption on traffic patterns in Tiara design, the traffic is generated in a random manner.

5.1 Micro-benchmarks

A few micro-benchmarks are designed to evaluate the major component optimizations described in §3.3. Specifically, we evaluate the lookup performance of our hash table design, measure the insertion speed of offloading engines, and test the PCIe overhead incurred by our aging mechanism.

Tiara OCT provides line-rate lookup. We run a benchmark with 10M flows in the OCT, implemented with three candidate hash table structures described in §3.3.1, i.e., 32M×1,

16M×2, and 8M×4. Figure 9 shows the lookup throughput on 10M flows with 128-byte packet size in three candidate hash structures. It reveals that both 32M×1 and 16M×2 structures approach line rate (97.2 Gbps and 97.15 Gbps), but 16M×2 provides a lower theoretic hash collision rate. When the width expands to 4, the throughput drops to 72.9 Gbps since all channels are used for each access at this width. This benchmark is consistent with our analysis in §3.3.1.

Tiara offloading engine achieves fast entry offloading. We randomly generate new flow entries in SMuxes and offload them to T-NIC by offloading engines. Therefore, in this experiment, all offloading operations are entry insertions. Figure 10 demonstrates the offloading speed of a single offloading engine, which is shared among SMuxes. The speed sticks to 6.8M operations per second with more than two SMuxes, which is bounded by the offloading engine. Figure 11 further shows how offloading speed changes with more offloading engines working in parallel. It achieves near linear-scaling with $2.77\times$ speedup when using three offloading engines. The linear scalability of offloading speed makes Tiara able to support a high CPS scenario. For example, the LB in [5] processes 6.9M CPS, requiring at least 13.8M offloading operations (insertion or deletion), which can be supported by two offloading engines, as shown in Figure 11.

Tiara entry aging mechanism incurs negligible overhead. We evaluate the PCIe utilization caused by the aging mechanism, i.e., sending signals (packet headers) to SMuxes via PCIe, with 10M flows and a 1 minute detection period. The average PCIe utilization is less than 0.05%. Given that the control plane and data plane share the same PCIe interface, the low PCIe utilization of Tiara aging mechanism incurs little influence on the data plane.

5.2 Tiara Performance

A complete Tiara system consists of at least one T-switch connected by multiple T-servers, each hosting up to 8 200GE T-NICs. We will show in this section whether such a system could meet the design goals: > 1 Tbps, > 10M concurrent flows, and > 1M CPS, without any assumption on traffic pat-
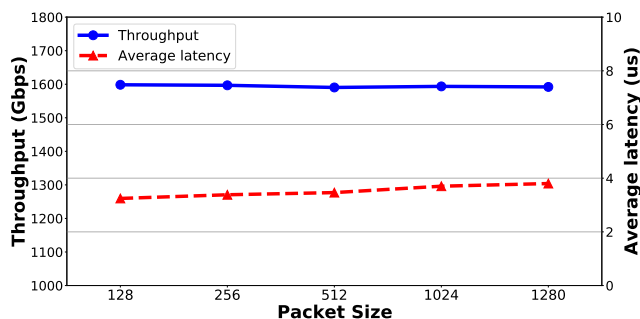
Figure 12: Forwarding performance in Tiara fast path. A T-server with 8 T-NICs achieves up to 1.6 Tbps with less than 4 us latency.

terns.

Throughput and latency. To measure the throughput and latency of Tiara with 10M concurrent flows, we generate traffic consisting of 10M flows and send them to Tiara, which offloads these flows into the fast path.

We first test the performance of Tiara fast path with a single T-NIC. It can achieve the line rate of 200 Gbps and provide an extremely low average latency of less than 4 us, with packet sizes ranging from 128 to 1280 bytes. We further break down the latency distribution in Tiara fast path, which shows about 1 : 1 latency between T-switch and T-NIC.

The throughput and the number of concurrent flows supported in one T-server can scale linearly with the number of T-NICs, as T-NICs plugged in the same server are totally independent of each other, and they share nothing in the fast path processing. As a result, with 8 T-NICs in one T-server, the aggregate throughput of T-server fast path scales linearly to 1.6 Tbps, and the latency remains exactly the same as that of a single T-NIC (i.e., less than 4 us), as shown in Figure 12. Similarly, the number of concurrent flows increases to 80M for a holistic T-server with 8 T-NICs. If the throughput requirement of an LB system is larger than 1.6 Tbps or the flow number requirement is larger than 80M, more T-servers can be connected to the T-switch tier, given the flexibility of this architecture. The aggregate throughput and the flow capacity of Tiara in the fast path can also scale linearly with the number of T-servers, as they are physically independent as well.

CPS. We evaluate Tiara ability to serve new TCP connections by issuing HTTP transactions, including a TCP connection establishment, an HTTP GET request, an HTTP response (by-passing LB), and a closure of TCP connection. We gradually increase the target CPS in 0.1M granularity at the generator to find the maximum available CPS that the target LB can serve all the incoming requests. The result reveals that Tiara can support up to 1.8M CPS (bounded by SMux), which is higher than our goal (i.e., 1M CPS).

Resilience to traffic patterns. By leveraging the large capacity of FPGA HBM for the connection table, almost all
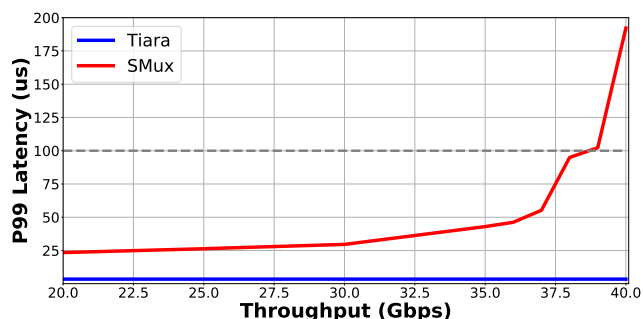

Figure 13: Latency-bounded throughput. With the tail (P99) latency bound of 100us, Tiara can achieve 200Gbps per T-NIC and 1.6Tbps per T-server with 8 T-NICs. However, since SMux suffers from high jitter when the load increases, the maximum latency-bounded throughput of SMux is 38Gbps.

flows can be offloaded to the fast path in Tiara as long as the number of concurrent flows is less than 10M per T-NIC and 80M per T-server, which is true in most cases as we observed at our datacenter boundaries. As a result, Tiara is insensitive to traffic patterns, and it keeps consistent high throughput and low latency on different traffic patterns.

5.3 Tiara vs. Existing Approaches

In this section, we compare Tiara with existing approaches (the SMux baseline, Silkroad [31]) in terms of performance and efficiency. The results are summarized in Table 1.

Performance. SMux suffers from high latency and jitter when the traffic load is heavy [33] due to high CPU utilization and cache misses. High latency and jitter will adversely impact the user's network experience. Therefore, we use "latency-bounded throughput" as the metric to compare SMux and Tiara more fairly. Given that the end-to-end latency from Internet users to datacenter services could be as low as a few milliseconds [35, 39], we should bound the tail latency of LB to the sub-millisecond level to minimize its impact on the user's network experience. In this experiment, we run SMux on 16 cores of a server with the same configuration as T-server (§4.3), except that the SMux server is equipped with a 100 Gbps Mellanox ConnectX-5 NIC rather than T-NICs. We set the bound of LB P99 latency to 100 us and compare the latency-bounded throughput with the packet size of 512 bytes between Tiara and SMux. Figure 13 shows the P99 latency of Tiara and SMux, respectively, with different throughputs. For the Tiara fast path, P99 latency is consistently below 4 us at throughput up to 200 Gbps per T-NIC, while SMux P99 latency breaks the 100 us bound when the throughput is higher than 38 Gbps. Therefore, we consider 38 Gbps as the maximum latency-bounded throughput of the baseline SMux. In Tiara, the latency-bounded throughput of a single T-server with 8 T-NICs is 1.6 Tbps, 42.1× higher than SMux (38 Gbps), and its P99 latency (4 us) is 25× lower than

	Throughput	P99 lat.	CPS	CT size*	Cost efficiency	Energy efficiency	Space efficiency
SMux	38 Gbps	100 us	1.8M	~100 GB	4.75 Gbps/(cost unit)	76 Mbps/Watt	19 Gbps/U
Silkroad**	1.6 Tbps	< 2 us	200K	100 MB	457.14 Gbps/(cost unit)	2909.1 Mbps/Watt	1600 Gbps/U
Tiara	1.6 Tbps	< 4 us	1.8M	4 GB	82.05 Gbps/(cost unit)	969.7 Mbps/Watt	320 Gbps/U

Table 1: Performance and efficiency comparison among different LBs. *Since the connection table (CT) compression in Silkroad is orthogonal to Tiara and can be applied in any architecture, we use the CT size as the metric to compare the data plane scalability of different architectures. **The Silkroad paper does not report throughput and tail latency explicitly, and we use the same throughput and latency results as T-switch to simplify comparison.

SMux (100 us).

Silkroad achieves comparable high throughput and low latency as Tiara, as most connections are processed in the hardware fast path in both solutions. However, Silkroad is less scalable in both control and data paths than Tiara. Silkroad leverages the embedded management CPU in switch for connection creation and offloading, thus expecting only 200K CPS [31]. Tiara achieves 1.8M CPS, 9× higher than Silkroad, thanks to the optimizations in the control plane of Tiara. Silkroad stores the connection table in the switch's limited on-chip SRAMs. Despite compression with hash digest, the connection table is still bounded by the on-chip SRAM size, i.e., 50-100 MB in modern switching ASICs. Tiara leverages 4 GB HBM in modern FPGA, increasing the connection table size in the fast path by orders of magnitude compared to Silkroad.

Efficiency. In this section, we quantify and compare the efficiency of SMux, Silkroad, and Tiara, in terms of cost efficiency (performance per dollar), energy efficiency (performance per watt), and space efficiency (performance per rack unit).

- **Cost efficiency.** As the concrete cost numbers of T-NIC, T-switch, and T-server used in the Tiara prototype are confidential, we normalize them to 1, 3.5, and 8, respectively. With these cost units, the normalized system costs of SMux, Silkroad, and Tiara are 8, 3.5, and 19.5 (=3.5+1*8+8), respectively. Given these normalized system costs and the throughput data shown in Table 1, the cost efficiency of these three approaches will be 4.75 Gbps/(cost unit), 457.14 Gbps/(cost unit), and 82.05 Gbps/(cost unit), respectively.

- **Energy efficiency.** According to hardware datasheets, T-NIC, T-switch, and T-server used in the Tiara prototype consume 75 Watt, 550 Watt, and 500 Watt power, respectively. Based on these power consumption and throughput data, the energy efficiency of SMux, Silkroad, and Tiara will be 76 Mbps/Watt, 2909.1 Mbps/Watt, and 969.7 Mbps/Watt, respectively.

- **Space efficiency.** The server used in SMux is 2 rack-unit (i.e., 2U) high, the switch used in Silkroad is 1U high, and the entire Tiara system is 5U high, as it includes a 1U T-switch and a 4U T-server hosting 8 T-NICs. Based on these

heights and throughput data, the space efficiency of SMux, Silkroad, and Tiara will be 19 Gbps/U, 1600 Gbps/U, and 320 Gbps/U, respectively.

- **Tiara vs. SMux in efficiency.** The cost, energy, and space efficiency of Tiara are 17.4×, 12.8×, and 16.8× higher than those of SMux, respectively. In other words, given the same target throughput, Tiara costs 17.4× less money, consumes 12.8× less energy, and takes 16.8× less rack space than SMux. All these efficiency advantages of Tiara over SMux come from hardware acceleration, as suitable hardware (i.e., FPGA and programmable switch in Tiara) is fundamentally much more efficient than x86 servers in network packet processing.

- **Tiara vs. Silkroad in efficiency.** As we can see from Table 1, the switch-only solution in Silkroad outperforms Tiara in all efficiency metrics. This is expected as Silkroad only leverages a switch, which is fundamentally more cost-, energy- and space-efficient than FPGA and x86 in network packet processing. However, as we discussed in the above section, the efficiency of Silkroad comes at the cost of lower CPS and smaller connection tables due to switch inherent scalability limitations. Compared to Silkroad, Tiara strikes a better balance between efficiency and scalability. Furthermore, the switch-only solution may not be that practical in traffic scenarios with a large number of connections, where Silkroad suggests operators combine its switch with an SMux for the slow path [31]. With this hybrid setting (switch + server), the efficiency of Silkroad will become similar to Tiara, but its scalability in hardware is still lower than Tiara.

One more option to further improve the efficiency of Tiara is to bake its implementation into a custom ASIC, which makes it as efficient as the Silkroad switch-only solution and as scalable as current Tiara. However, a custom ASIC incurs a significant NRE (non-recurring engineering) cost. Without a big enough volume to amortize the NRE, the cost efficiency of custom ASIC is worse than that of current Tiara design. As the performance of a single T-server is already high enough (up to 1.6 Tbps), we do not necessarily need a large number of T-servers to load-balance Internet traffic in even hyperscale datacenters. Therefore, the design choice of using FPGA rather than custom ASIC in Tiara is justified in this context.

6   Related Work

Memory enhanced switches: eXtra Large Table (XLT) [17] enhances programmable switches with FPGA + DRAM complexes to support large tables. It works well when all rule/flow tables are stored in DRAM, and the switch and FPGA can handle all data plane processing entirely. However, that is not the case for stateful load balancers discussed in this paper. Despite large DRAM, packet lookup may still miss in XLT FPGA due to hash collision or first packet processing for new connections, but how to handle these exceptions is unclear.

TEA [25] extends switching ASIC memory virtually by utilizing the host DRAM via RDMA. However, looking up a table at the remote memory prevents switches from line-rate processing. TEA relies heavily on traffic locality that caches hot traffic in the on-chip SRAMs to preserve high throughput. Otherwise, its performance approaches the server-based lookup table, as demonstrated in its experiment (TEA with and without cache). Moreover, TEA shares the same scalability issue on the control plane as other programmable switches.

Layer-4 load balancing: There have been continuous efforts on layer-4 load balancing. In general, two LB categories are explored: stateful LBs that keep the per-connection state at Muxes and stateless LBs that do not maintain any per-connection state.

Ananta [36] and Maglev [21] are two proposed software stateful LBs with a series of packet processing optimizations, including batch processing, poll mode NIC driver, and zero-copy operations. Despite these optimizations, the packet forwarding throughput on a single server is still limited, so that they need a large number of servers to support terabits per second traffic.

Duet [24] and Rubik [23] accelerate Ananta with commodity switches in a stateless style. They store the VIP-to-DIP (RS_IP) mapping in switch on-chip SRAMs as an ECMP table. To support large-scale mapping rules, they leverage the tail distribution in VIP traffic to configure the heavy-hitting rules on switches while processing the rest in the software.

Beamer [33] is a recently proposed stateless LB. It relies on hash functions on the switch to proceed fast real server selection and uses "daisy chaining" techniques to mitigate the PCC violations. The "daisy chaining" requires real servers to redirect unexpected packets. However, it is empirically impractical to modify the service servers. Moreover, stateless LBs can only provide suboptimal workload balancing due to the nature of hash functions as described in [16].

Silkroad [31] is the most related work, which accelerates stateful LB with programmable switches. It faces the same problems as mentioned in §2.3, but it only focuses on addressing the data plane scalability issue with on-chip SRAMs. Silkroad stores a hash digest of a connection instead of the 5-tuple in the connection table, which reduces the key size

of each connection from dozens of bytes to 16 bits. Such compression technique scales to support millions of concurrent flows. However, Silkroad will suffer from throughput degradation due to pipeline folding for those switches that distribute their SRAM resources in multiple pipelines.

Cheetah [16] aims to design a high-speed LB for both stateless and stateful manners. One of its contributions is to solve the entry insertion inefficiency problem in stateful LB by storing unused hash indexes in a connection stack. For every new coming connection, Cheetah pops an index from the connection stack and inserts the connection entry into the hash table with the retrieved index. This index, encoded in the packet header as a cookie, is carried by the connection in the following packets. The change on the packet header requires modifications on services' client sides. This requirement prevents Cheetah from deploying on large-scale datacenters with hundreds and thousands of services.

Component design & optimization: Some techniques used in the component design and optimization in Tiara have been extensively studied. Tong et al. [41] propose a high-throughput hash table structure with the idea of fixed-length hashing in FPGA DRAM. Mogul et al. [32] eliminate the livelock by a polling-based mechanism, and Kuperman et al. [27] match each net device TX queue to a hardware send queue to avoid spin-lock contention. Ross [38] splits tables into different cores in a multi-core database system to reduce the synchronization cost. The SmartNIC used in Azure [22] periodically reports flow states to the software, which allows the software manager to age the inactive flows. Our contribution is to integrate those techniques to achieve the design goals of Tiara.

7   Conclusion

Tiara is a novel hardware acceleration architecture for stateful load balancers. It simultaneously provides high throughput, low latency, high scalability, and high efficiency by mapping different LB tasks into their most suitable hardware and carefully designing and optimizing a few key components. Although we only show Tiara's capabilities to accelerate stateful load balancers in this paper, we believe this architecture is generic for network function acceleration and can be explored in the future in more gateway scenarios, such as DDoS protection and firewall.

Acknowledgments

References

[1] Axi high bandwidth memory controller v1.0. https://www.xilinx.com/support/documentation/ip_documentation/hbm/v1_0/pg276-axi-hbm.pdf.

[2] Dpdk. https://www.dpdk.org/.

[3] Dpvs is a high performance layer-4 load balancer based on dpdk. https://github.com/iqiyi/dpvs.

[4] express data path. https://www.iovisor.org/technology/xdp.

[5] High-performance dpdk-based server load balancing for alibaba singles' day shopping festival. https://www.alibabacloud.com/blog/593984?spm=a2c5t.11065265.1996646101.searchclickresult.289b2f059llA1a.

[6] Ieee 802 numbers. https://www.iana.org/assignments/ieee-802-numbers/ieee-802-numbers.xhtml.

[7] Intel intrinsics guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide.

[8] Katran: A high performance layer 4 load balancer. https://github.com/facebookincubator/katran.

[9] Load balancing 101: Nuts and bolts. https://www.f5.com/services/resources/glossary/load-balancer.

[10] Ovs conntrack. https://docs.openvswitch.org/en/latest/tutorials/ovs-conntrack/.

[11] Qdma subsystem for pci express. https://www.xilinx.com/products/intellectual-property/pcie-qdma.html.

[12] Unveiling the networks behind the 2018 double 11 global shopping festival. https://www.alibabacloud.com/blog/594167?spm=a2c5t.11065265.1996646101.searchclickresult.289b2f0575gg5Z.

[13] Virtual extensible local area network (vxlan): A framework for overlaying virtualized layer 2 networks over layer 3 networks. https://tools.ietf.org/html/rfc7348.

[14] Anurag Agrawal and Changhoon Kim. Intel tofino2–a 12.9 tbps p4-programmable ethernet switch. In *HCS 2020*.

[15] João Taveira Araújo, Lorenzo Saino, Lennert Buytenhek, and Raul Landa. Balancing on the edge: Transport affinity without network state. In *NSDI 2018*.

[16] Tom Barbette, Chen Tang, Haoran Yao, Dejan Kostić, Gerald Q Maguire Jr, Panagiotis Papadimitratos, and Marco Chiesa. A high-speed load-balancer design with guaranteed per-connection-consistency. In *NSDI 2020*.

[17] Curt Beckmann, Ramkumar Krishnamoorthy, Han Wang, Andre Lam, and Changhoon Kim. Hurdles for a dram-based match-action table. In *ICIN 2020*.

[18] John Black, Shai Halevi, Hugo Krawczyk, Ted Krovetz, and Phillip Rogaway. Umac: Fast and secure message authentication. In *CRYPTO 1999*.

[19] Peter Bodík, Ishai Menache, Mosharaf Chowdhury, Pradeepkumar Mani, David A Maltz, and Ion Stoica. Surviving failures in bandwidth-constrained datacenters. In *SIGCOMM 2012*.

[20] Alan Edelman. Akamai technologies: A mathematical success story. In *SIAM News 1999*.

[21] Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *NSDI 2016*.

[22] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure accelerated networking: Smartnics in the public cloud. In *NSDI 2018*.

[23] Rohan Gandhi, Y Charlie Hu, Cheng-Kok Koh, Hongqiang Harry Liu, and Ming Zhang. Rubik: unlocking the power of locality and end-point flexibility in cloud scale load balancing. In *ATC 2015*.

[24] Rohan Gandhi, Hongqiang Harry Liu, Y Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. Duet: Cloud scale load balancing with hardware and software. In *SIGCOMM 2014*.

[25] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. Tea: Enabling state-intensive network functions on programmable switches. In *SIGCOMM 2020*.

[26] Hugo Krawczyk. Lfsr-based hashing and authentication. In *CRYPTO 1994*.

[27] Yossi Kuperman, Maxim Mikityanskiy, and Rony Efraim. Hierarchical qos hardware offload (htb).

[28] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *SOSP 2017*.

[29] Bojie Li, Kun Tan, Layong Luo, Yanqing Peng, Ren-qian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *SIGCOMM 2016*.

[30] Yishay Mansour, Noam Nisan, and Prasoon Tiwari. The computational complexity of universal hashing. In *TCS 1993*.

[31] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *SIGCOMM 2017*.

[32] Jeffrey C Mogul and KK Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. In *TOCS 1997*.

[33] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. Stateless datacenter load-balancing with beamer. In *NSDI 2018*.

[34] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *Journal of Algorithms 2004*.

[35] Fabio Palumbo, Giuseppe Aceto, Alessio Botta, Domenico Ciuonzo, Valerio Persico, and Antonio Pescapé. Characterization and analysis of cloud-to-user latency: the case of azure and aws. In *CN 2021*.

[36] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, et al. Ananta: Cloud scale load balancing. In *SIGCOMM 2013*.

[37] Kun Qian, Sai Ma, Mao Miao, Jianyuan Lu, Tong Zhang, Peilong Wang, Chenghao Sun, and Fengyuan Ren. Flexgate: High-performance heterogeneous gateway in data centers. In *APNet 2019*.

[38] Kenneth A Ross. Multicore processors and database systems: The multicore transformation. In *Ubiquity 2014*.

[39] Ao-Jan Su, David R Choffnes, Aleksandar Kuzmanovic, and Fabian E Bustamante. Drafting behind akamai: Inferring network conditions based on cdn redirections. In *TON 2009*.

[40] Ao-Jan Su and Aleksandar Kuzmanovic. Thinning akamai. In *SIGCOMM 2008*.

[41] Da Tong, Shijie Zhou, and Viktor K Prasanna. High-throughput online hash table on fpga. In *IPDPS 2015*.

[42] Teng Zhang, Jianying Wang, Xuntao Cheng, Hao Xu, Nanlong Yu, Gui Huang, Tieying Zhang, Dengcheng He, Feifei Li, Wei Cao, et al. Fpga-accelerated compactions for lsm-based key-value store. In *FAST 2020*.

A Analysis on Hash Collision

Suppose there are n random entries inserted into a hash table with width w and depth d. The probability that any i entries are hashed to the same index is:

$$p(i) = C_n^i (\frac{1}{d})^i (1 - \frac{1}{d})^{n-i} \qquad (1)$$

The probability for any indexes that hold $0 \sim w$ entries is:

$$p(num \leq w) = \sum_{i=0}^{w} C_n^i (\frac{1}{d})^i (1 - \frac{1}{d})^{n-i} \qquad (2)$$

For all indexes, this probability becomes:

$$p(num \leq w)_{all} = (\sum_{i=0}^{w} C_n^i (\frac{1}{d})^i (1 - \frac{1}{d})^{n-i})^d \qquad (3)$$

Therefore, the probability for all indexes that exist at least once collision, i.e., holding more than w entries, is:

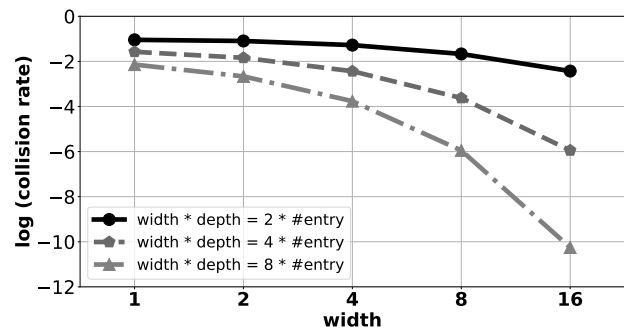$$p(num > w)_{all} = 1 - (\sum_{i=0}^{l} C_n^i (\frac{1}{d})^i (1 - \frac{1}{d})^{n-i})^d \qquad (4)$$

Figure 14: The numerical simulation on collision rates in different widths and depths with #entry $= 32768$. The collision rates are shown in the log scale.

To get an intuitive relationship between the collision rate and the width, we conduct a numerical simulation on different settings based on Equation 4. The results are demonstrated in Figure 14, and show that given a fixed hash space ($> n$), a larger width results in a lower hash collision rate.

Scaling Open vSwitch with a Computational Cache

Alon Rashelbach, Ori Rottenstreich, Mark Silberstein
Technion

Abstract

Open vSwitch (OVS) is a widely used open-source virtual switch implementation. In this work, we seek to scale up OVS to support hundreds of thousands of OpenFlow rules by accelerating the core component of its data-path - the packet classification mechanism. To do so we use NuevoMatch, a recent algorithm that uses neural network inference to match packets, and promises significant scalability and performance benefits. We overcome the primary algorithmic challenge of the slow rule update rate in the vanilla NuevoMatch, speeding it up by over *three orders of magnitude*. This improvement enables two design options to integrate NuevoMatch with OVS: (1) using it as an extra caching layer in front of OVS's megaflow cache, and (2) using it to completely replace OVS's data-path while performing classification directly on OpenFlow rules, and obviating control-path upcalls. Our comprehensive evaluation on real-world packet traces and ClassBench rules demonstrates the geometric mean speedups of $1.9\times$ and $12.3\times$ for the first and second designs, respectively, for 500K rules, with the latter also supporting up to 60K OpenFlow rule updates/second, by far exceeding the original OVS.

1 Introduction

Open vSwitch (OVS) [22] is one of the most popular software switches used by cloud providers to implement software-defined networks [1, 8, 23]. As part of its main tasks, OVS classifies packets according to a set of match-action tuples, i.e., OpenFlow rules dynamically installed by the network controller. To achieve high throughput, OVS adopts the fast/slow path separation principle: the majority of the packets are classified in the fast *data-path*, which maintains a *megaflow cache* optimized for speedy matching. Upon a miss, OVS invokes the slower upcall into a *control-path*, which populates the megaflow cache with tuples called megaflows.

Unfortunately, OVS suffers from two primary scalability issues. First, the megaflow cache becomes slower as the number of megaflows in it grows. Our experiments (§3) show that with 500K megaflows, OVS is about an order of magnitude slower than with 1K megaflows. Importantly, the cache might hold a large number of megaflows even if the number of the original OpenFlow rules is small. This is because when OVS populates the cache, it transforms the relevant OpenFlow rules into a set of non-overlapping megaflows [22]. As a result, the OpenFlow rules might get fragmented; under certain common traffic patterns, this fragmentation leads to a dramatic increase in the number of megaflows in the cache [3, 4].

The second problem is the performance degradation that occurs when new rules are inserted into OVS by a network controller. We observe (§3) that the throughput might be affected significantly even when adding only a few dozens of new OpenFlow rules at a time. The main reason stems from the need to enforce the non-overlapping property of megaflows, which might cause OVS to remove existing megaflows, leading to slow path upcalls. Clearly, the problem gets worse in systems with frequent rule updates.

In this work, we seek to overcome these OVS limitations. Our key idea is to leverage the recently published algorithm for packet classification, called *NuevoMatch* [26, 27], which was shown to significantly outperform state-of-the-art alternatives when scaling to a large number of OpenFlow rules. NuevoMatch uses shallow neural networks comprising a *Range-Query Recursive Model Index* (RQ-RMI) to learn the distribution of the rules. The rule lookup is translated into neural-network inference that replaces the traditional index data structure traversal. Upon an update, new rules are first added to a slow-path *remainder* classifier, and the model is periodically retrained to incorporate them in the fast path. Thus, the RQ-RMI model serves as a *computational cache* for the remainder, while retraining the model is equivalent to filling that cache. The scalability of NuevoMatch follows from its small memory footprint and efficient use of CPU hardware, which together enable fast execution on modern CPUs [26].

However, our initial attempts to integrate OVS and Nuevo-Match revealed one critical limitation of the original algorithm: its inability to accommodate fast updates. When rules are modified, NuevoMatch must *retrain the RQ-RMI model*

from scratch on the updated rule-set, in order to reach its full performance potential. Unfortunately, RQ-RMI training time is too long and cannot support the required update rate, particularly with a large number of OpenFlow rules as targeted by our work. Our analysis (§3) shows that the NuevoMatch training rate is *orders of magnitude* slower than the one necessary to achieve its promised performance benefits.

We tackle this challenge by introducing *NuevoMatchUP* which extends the original NuevoMatch training algorithm and improves the training rate by over *three orders of magnitude*. Thus, it requires only a few milliseconds to train tens of thousands of rules, and about one second for 500K rules, thereby paving the way to the practical integration of computational cache into OVS.

We consider two design options for integrating Nuevo-MatchUP with OVS. The first design, *OVS with computational cache (*OVS-CCACHE*)*, targets the scalability of the megaflow cache by accelerating it with NuevoMatchUP. OVS-CCACHE achieves higher throughput than the original design, but unfortunately inherits the low rule update performance. To support fast updates, we introduce *OVS with computational flows (*OVS-CFLOWS*)*, which leverages the power of Nuevo-MatchUP to efficiently match complex OpenFlow rules and obviates the need for the megaflow cache and fast-slow path separation of the original OVS. This change eliminates the the key bottleneck that restricts the rule update rates in the original OVS.

We comprehensively evaluate OVS-CCACHE and OVS-CFLOWS using real-world CAIDA [2] and MAWI [37] traces, and the standard ClassBench-generated rule-sets [32]. OVS-CCACHE improves the megaflow cache performance, achieving the end-to-end geometric mean speedups of $1.5\times$, and $1.9\times$ for 100K, and 500K OpenFlow rules, respectively.

OVS-CFLOWS sidesteps the control-path limitations and is thus significantly faster, with the end-to-end geometric mean speedups of $2.6\times$, $8.5\times$, and $12.3\times$ for 1K, 100K, and 500K OpenFlow rules, respectively. Moreover, OVS-CFLOWS handles more than 60K OpenFlow rule updates/second.

These results demonstrate the first practical use of RQ-RMI models in a production packet processing system, and show their ability to improve throughput and scalability.

2 Background

We explain the relevant details about the operation of Open vSwitch (OVS) [22, 23], and describe the NuevoMatch algorithm [26] for packet classification.

2.1 Open vSwitch

Open vSwitch (OVS) is a popular open-source virtual switch that supports industry standard OpenFlow protocols. OVS determines which action to apply on each packet according to the OpenFlow rules installed by the network controller.
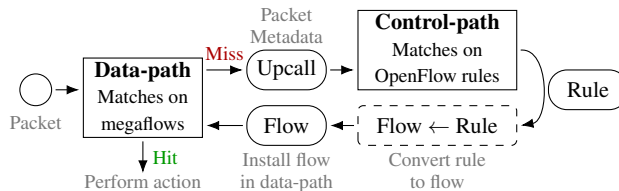

Figure 1: Fast/slow path separation in OVS.

This task is known as *packet classification*, and has been extensively studied [6, 10, 18, 19, 26, 28, 30, 35, 39].

Matching a rule. In its simplest form, a *rule* is a boolean predicate parametrized by one or more fields in the packet header (e.g., IP address, IP protocol). If a predicate is true for a given packet (the rule *matches*), an *action* associated with the rule is invoked to process the packet. An action is an operation to apply to the packet (e.g., forward to port or drop). A packet may match several *overlapping* rules, but only the one with the highest priority is selected.

Control-/data- path. OVS is split into data- (fast) and control-(slow) paths (Figure 1). All OpenFlow rules are installed and maintained in the control-path. The data-path, on the other hand, uses a *megaflow cache* to achieve high processing rates.

The megaflow cache holds non-overlapping rules called *megaflows*, generated by the control-path from the installed OpenFlow rules. Specifically, whenever the data-path encounters a packet that does not match any previously installed megaflow, it performs an *upcall* to the control-path, which in turn finds the relevant OpenFlow rule and converts it into a megaflow. Future packets with the same header fields will not require upcalls unless the megaflow is removed. OVS ensures the correctness of the matching process with the megaflow cache, terminating lookup after a hit in it. To achieve that, OVS tracks all modifications to the OpenFlow rules in the control-path. In particular, it might need to invalidate previously installed megaflows when new OpenFlow rules are added. As we show in §3, these operations might significantly affect OVS's performance.

In addition to the megaflow cache, OVS often activates a short-term exact-match cache (EMC) in front of it. The EMC can be helpful with high-locality traffic.

Megaflow cache implementation. The megaflow cache uses the *Tuple Space Search* (TSS) [30] algorithm for packet classification, as follows. Megaflows with the same mask m are stored in the same hash table H_m, with masked flow keys as entries. Given a packet header h, the megaflow cache iterates over all hash tables to find an entry that matches h (i.e., the masked header equals to the masked key). The lookup latency increases linearly with the number of hash tables traversed.

OVS's data-path can run either in the user-space using DPDK [25], or as a dedicated Kernel module. In this paper we use the DPDK version for its higher performance [34].
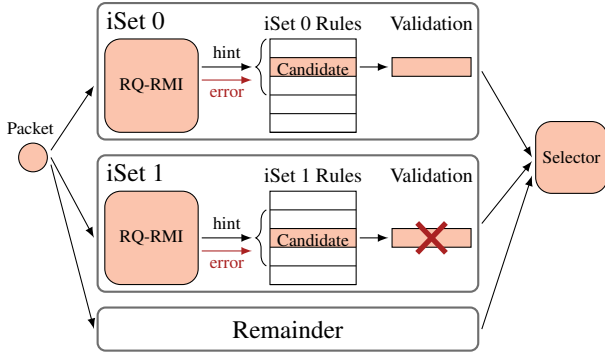
Figure 2: NuevoMatch algorithm [26], RQ-RMI inference provides hints to find the matching rule (details in §2.2).

Size Category	SC 0	SC 1	SC 2	SC 3
# Input Rules	$< 10^3$	10^3 - 10^4	10^4-10^5	$> 10^5$
# Neural Nets	5	21	133	265 or 521

Table 1: RQ-RMI model size (number of neural nets) for different number of rules to index (values taken from [26]).

2.2 NuevoMatch Classification Algorithm

NuevoMatch (NM) is a new class of packet classification algorithms that leverage neural nets to scale to many rules [26].

Figure 2 presents the main components of the algorithm. NM partitions a given set of rules into several independent subsets (iSets), such that each iSet s has a header field h_s in which its rules do not overlap. The fraction of an iSet's rules out of all rules is called the *iSet's coverage*. In practice, two iSets are often sufficient to cover more than 90% of the rules for large enough rule-sets [26]. Rules that do not fit in any of the iSets are handled by a *remainder classifier*, which can be implemented by any other packet classification technique.

For each iSet s, NM trains a hierarchical model called *Range-Query Recursive Model Index* (RQ-RMI) which consists of multiple shallow neural-nets. RQ-RMI learns the distribution of ranges represented by the rules and outputs the *estimated* index of the matching rule within an array. At the inference time, this estimation is used as a starting index to *search* for the matching rule within the array. Crucially, the RQ-RMI training algorithm guarantees a tight bound on the maximum error of the estimated index, which in turn bounds the search and ensures lookup correctness. During the search, the candidate rules are *validated* by matching over *all* fields of the incoming packet. Finally, the highest priority rule is selected out of all the matching rules from all the iSets and the remainder.

The number of neural nets (NNs) in an RQ-RMI model depends on the number of rules it indexes. The original paper suggests four RQ-RMI size categories, reported in Table 1. The larger the model, the longer it takes to train it. However
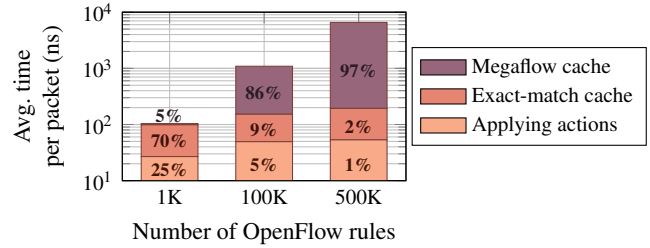

Figure 3: Breakdown of packet processing times in the data-path for different number of OpenFlow rules.

smaller models would fail to achieve the target error bound guarantees and would result in a slower lookup. Thus, there is a fundamental trade-off between the lookup latency and the training time. NuevoMatchUP changes the way RQ-RMI is constructed to modify this trade-off, allowing a much faster training with negligible degradation in the lookup latency.

3 Motivation

We analyze OVS's scalability bottlenecks and highlight the potential benefits of using faster packet classification.

For the analysis we use the same setup and workloads as described in §7. In particular, we generate 36 ClassBench Open-Flow rule-sets (12 application types of three size categories each: 1K, 100K, 500K rules), and evaluate the throughput by replaying Caida-short packet trace (100M packets).

Does OVS get slower with more rules? We compare the throughput with 1K rules vs. the throughput with 100K and 500K rules, separately for each ClassBench application type. We observe that the geometrical mean *slowdown* for 100K and 500K rules vs. 1K rules is 5.8× and 9.1×, respectively.

Takeaway 1: OVS does not scale well to a large number of OpenFlow rules.

Where is the bottleneck in the data-path? We analyze the average processing time of a packet in the OVS data-path while varying the number of OpenFlow rules across all the rule-sets. Figure 3 shows that packets spend the majority of time in the megaflow cache, i.e., 86% and 97% of the CPU time on average, for 100K and 500K rules respectively.

Takeaway 2: OVS megaflow cache becomes the main data-path performance bottleneck as the number of OpenFlow rules increases.

Are the control-path upcalls the primary bottleneck? Misses in the megaflow cache trigger upcalls into the control-path. The frequency of the upcalls is hard to predict; it depends on the interplay between the rule-set and the traffic pattern [3, 4]. Unfortunately, frequent upcalls cause major throughput drop. For example, Figure 4 shows the throughput and the rate of deletions and upcalls, sampled every 100ms, for a 100K rules (rule-set number 2 in §7). The higher the
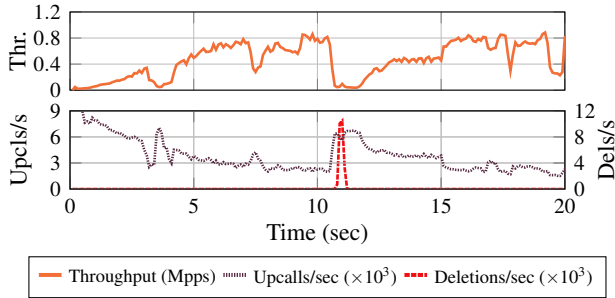
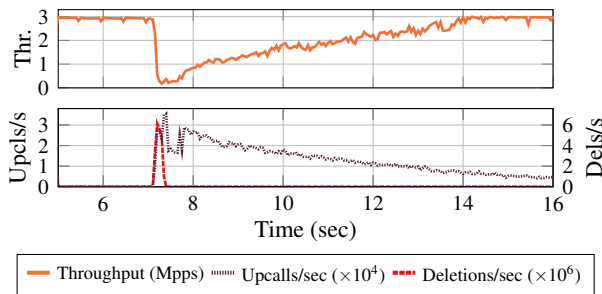Figure 4: OVS throughput is affected by control-path upcalls.

Figure 5: Insertion of new OpenFlow rules: the throughput drops at $t = 7s$ when 60 new OpenFlow rules are added to 500 existing ones. Note the coinciding peak in the deletion rate from the megaflow cache and the subsequent increase in the number of upcalls.

number of upcalls, the lower the throughput. Similarly, the performance drop is observed due to deletions, triggered by the periodic megaflow cache cleanup of idle flows. For other rule-sets the behavior is similar.

Takeaway 3: frequent upcalls are detrimental to performance.

Impact of OpenFlow rule updates. OVS might experience a sharp drop in throughput when OpenFlow rules are modified.

To show that, we install 500 OpenFlow rules in the beginning and update 60 rules at time $t = 7$. Figure 5 shows the results. The moment before the update occurs, there are 144K megaflows in the cache. We see that the update causes about 104K deletions from the megaflow cache, followed by tens of thousands of upcalls. As a result, the throughput drops dramatically and takes a few seconds to recover.

This graph illustrates a general problem rooted in the megaflow algorithm. When inserting new rules that overlap existing ones with lower priorities, OVS must delete all megaflows that correspond to the existing rules (§3). While the magnitude of the throughput degradation depends on the rules being updated, the issue is significant in particular with high update rates.

Takeaway 4: modifying a handful of OpenFlow rules might significantly affect the throughput because of the increase in upcalls.
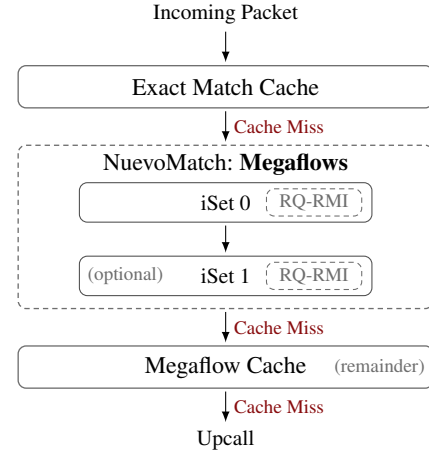
(a) OVS-CCACHE with NuevoMatch accelerating the megaflow cache.
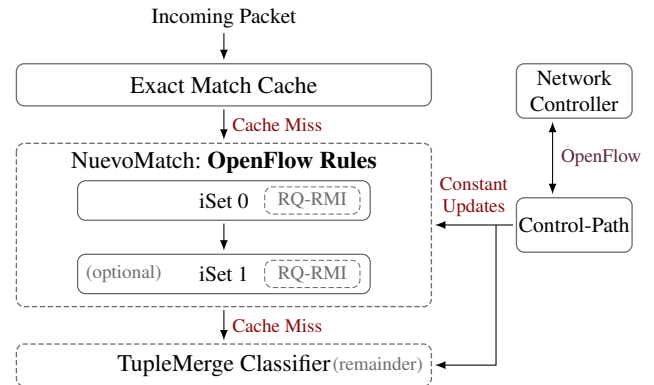
(b) OVS-CFLOWS with NuevoMatch performing OpenFlow rule classification in the data-path.

Figure 6: Design options for integrating NuevoMatch with OVS. See §8 for the discussion why to choose one over the other.

4 Design Options and Challenges

Our analysis indicates two primary reasons for the OVS performance degradation: (a) poor scalability of the megaflow cache; (b) frequent upcalls to the control-path. In the following we consider two designs to solve these issues.

4.1 OVS with Computational Cache

To tackle the first issue, the most natural solution is to replace the megaflow cache with a more scalable NuevoMatch. This approach is appealing because it fits well in the existing OVS design. Here, NuevoMatch uses the megaflow cache as a remainder, and can be seen as an additional layer of caching for megaflows. We call this approach an *OVS with a computational cache (OVS-CCACHE)*.

Figure 6a shows the proposed OVS-CCACHE design, depict-

Num. of OF rules	Upcalls per sec	Num. of Megaflows in cache	Training time est.	Coverage degrad. est.
1K	128	6.5K	30s	3.8%
100K	6.7K	102K	270s	7%
500K	6.4K	90K	250s	8%

Table 2: Characterization of rule update rate requirements in the megaflow cache. NuevoMatch training should be at least $100\times$ faster to be applicable to the megaflow cache.

ing only the data-path. The control-path is unmodified. Incoming packets are first matched against the exact-match cache. A miss is then forwarded to the computational cache provided by NuevoMatch RQ-RMI models. The original megaflow cache serves the lookups which did not match in RQ-RMI. If missed again, the packet continues with the original OVS upcall mechanism.

When new megaflows are added to the data-path, they are first inserted into the original megaflow cache. The RQ-RMI model is periodically re-trained in a separate thread by pulling the added megaflows from the megaflow cache. When the training finishes, the old RQ-RMI models are replaced with the newly trained ones that already incorporate the new megaflows, and the megaflow cache is emptied.

Unfortunately, this solution inherits the performance limitations of the upcall mechanism, and thus would not scale well in case of frequent upcalls.

4.2 OVS with Computational Flows

To solve the issue of slow upcalls, one option is to apply NuevoMatch to the control-path classifier to speed up the handling of upcalls. Unfortunately, control-path tasks go well beyond OpenFlow rule matching, and it is unclear how to use NuevoMatch in this context. Specifically, the control-path effectively implements the algorithm for tracking and generating non-overlapping megaflows. This is the core of the control-path and it is tightly coupled with the rule matching logic. Thus, NuevoMatch is not suitable for control-path acceleration.

On the other hand, the excessive number of upcalls we observed stems primarily from the design choice to generate non-overlapping megaflows for the data-path. The fact that megaflows do not overlap is an essential feature in OVS design that allows fast-path performance optimizations, but it is also the one that deteriorates the throughput dramatically in case of frequent upcalls [3, 4].

Therefore, our proposed solution, *OVS with computational flows (*OVS-CFLOWS*)*, leverages NuevoMatch to perform efficient packet classification directly on complex OpenFlow rules, without resorting to non-overlapping megaflows. As a result, we remove the megaflow cache mechanism and the

associated control-path logic, and obviate the need for upcalls. This approach, while more intrusive than OVS-CCACHE, holds the promise to boost OVS performance both with and without OpenFlow rule updates. How it fairs against OVS-CCACHE is one of the questions we answer in our evaluation.

Figure 6b shows the design of OVS-CFLOWS. While it resembles OVS-CCACHE, the difference is that NuevoMatch here is used to match OpenFlow rules instead of megaflows as in OVS-CCACHE. Similarly to OVS-CCACHE, updates are first inserted into the remainder (we use TupleMerge [6] for its implementation), and RQ-RMI models are periodically retrained to accommodate them.

4.3 Challenge: Slow NuevoMatch Updates

Unfortunately, in practice, NuevoMatch cannot support either OVS-CCACHE or OVS-CFLOWS. Recall that rule modification in the classifier requires retraining all its RQ-RMI models from scratch with the new, modified set of rules (§2.1). Therefore, the rule update rate is bounded by the training time of the models, which in turn depends on the number of rules in the classifier rather than on the number of modified rules.

In the following, we analyze the update rate requirements for OVS-CCACHE and OVS-CFLOWS, and show that NuevoMatch is over two orders of magnitude slower than required.

Megaflow cache rule churn. To understand the training rate requirements for NuevoMatch in OVS-CCACHE, we analyze the rule churn rate in the megaflow cache. For each OpenFlow rule size category we measure (1) the average rate of upcalls, which is equivalent to the rate of updates in the megaflow cache (we count insertions only, as NuevoMatch supports deletions without retraining), and (2) the average number of megaflows in the cache, which dictates the NuevoMatch training time if it were used to accelerate the megaflow cache.

Table 2 shows that for larger rule-sets (100K, 500K) there are about 6.5K upcalls per second, and the megaflow cache holds about 100K megaflows. Thus, NuevoMatch would have to retrain the model with 100K rules every 150 μs. This is of course unrealistic: training a model of that size would require about 270 seconds according to the original paper.

The solution suggested by the authors of NuevoMatch is to accumulate the updates in the remainder and serve the queries from it while training. Thus, the *coverage* of the RQ-RMI model is lower during the training; hence, the performance is lower because more queries are served in the remainder. When the training is finished, the coverage improves, and a new round of training begins right away to catch up with the rules modified during the previous training round.

Unfortunately, this option is not practical either. If 6.7K rules get modified each second, the expected coverage degradation per second would be about 7% (see Table 2). If we accumulate the updates while training for 270 seconds, the coverage will become practically zero, nullifying the NuevoMatch performance benefits completely. For comparison, even to
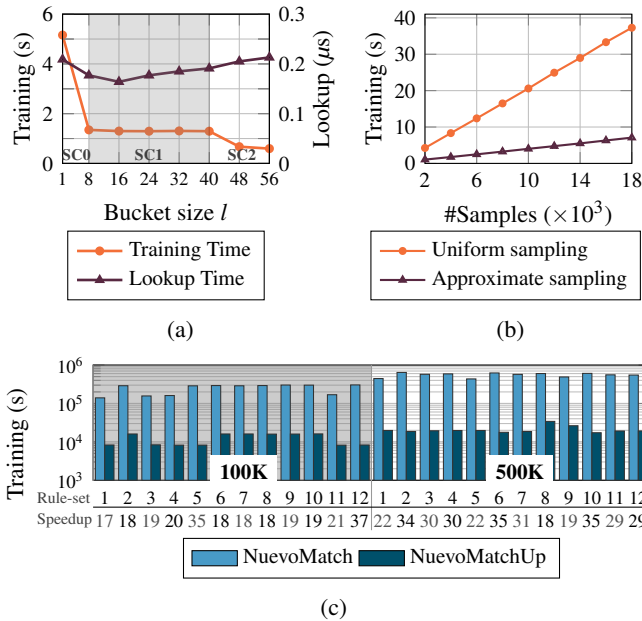
Figure 7: **(a)** The effect of the bucket size l on the RQ-RMI training and lookup times. See RQ-RMI size categories (SC) in Table 1. **(b)** Training using approximate sampling is faster. Here we train 500K rules using bucket size $l = 40$. **(c)** The training implementation of NuevoMatchUP is 20 times more efficient than that of NuevoMatch.

achieve the coverage of 25%, which is the cutoff suggested in the paper for NuevoMatch to provide minimum performance benefits, the training must complete within 4 seconds. This is almost *two orders of magnitude* faster compared to 270 seconds that NuevoMatch allows today. We use the same model as the original paper to produce these estimates: $E = R \cdot e^{-U/R}$, where R and U are the total number of rules and the number of updates respectively, and E is the expected number of rules in the model left after the updates.

OVS-CFLOWS update requirements. The update rate of Open-Flow rules varies between 400 to 338K updates per second [12, 13]. Supporting an average rate of 100K updates per second in NuevoMatch would require retraining every 500ms to achieve a coverage of 90% for a rule-set of 500K OpenFlow rules. Unfortunately, the actual NuevoMatch training time for a rule-set of that size is about 600s, which is over three orders of magnitude slower.

We conclude that *NuevoMatch training algorithm is too slow to support the update requirements of OVS in the considered designs*.

5 NuevoMatchUP: Speeding-up Updates

We introduce *NuevoMatchUP* (NMU), a series of enhancements to NuevoMatch which together significantly improve its update rate by several orders of magnitude.

NMU introduces important changes to the RQ-RMI construction and training algorithms, as well as to their implementation. First, it enables creating much smaller (thus faster-to-train) RQ-RMI models by constructing iSets with overlapping rules. Second, it enables major improvement in training speed by cutting down the number of memory accesses. We now discuss these changes in detail.

5.1 Relaxing iSet Constraints

An iSet s is a set of rules associated with a field h_s for which rules do not overlap (§2.2). We relax the no-overlap constraint, by allowing overlap between a certain number of rules. Informally, a *relaxed iSet* is an iSet with up to l overlapping rules in field h_s, grouped in *buckets* (defined next).

We now describe the algorithm for constructing a relaxed iSet s on a header field h_s.

Lemma 1. *Given an OVS classification rule r and a packet header field h, the set of values in h that match r can be represented by an integer range, denoted as $h(r)$.*

The correctness of the lemma directly follows from the usage of prefix based wildcard representation in OVS.

Definition 1. *A bucket is a set of up to l rules. We say that two buckets b_1 and b_2 do not overlap with respect to the header field h if for any rule r_1, r_2 in b_1, b_2 respectively, $h(r_1)$ does not overlap $h(r_2)$.*

To create buckets that do not overlap with respect to the header field h_s, we sort the rules by their ranges in h_s, and iterate over them allowing up to l overlapping ranges per bucket. Whenever we encounter a rule with a range that does not overlap with its predecessors, we include it in a new bucket. If buckets contain less than l rules, we merge adjacent buckets while keeping the constraint to have at most l rules per bucket. Next, we use RQ-RMI models to *learn the distribution of the buckets* rather than the distribution of the rules [26].

Since the number of buckets is smaller by up to a factor of l than the number of rules, RQ-RMI models in Nuevo-MatchUP are smaller and train faster than in NuevoMatch (see Table 1). Of course, the cost of this optimization is a slower lookup: all the rules in the same bucket must be validated via a linear scan. This trade-off, however, turned out to be beneficial to accelerate training with a negligible slow-down for the lookup. Figure 7a demonstrates this trade-off using a representative rule-set (12-500K, see §7). Buckets of sizes $l = 8, 48$ change the RQ-RMI size category and dramatically improve the model's training performance. Other bucket sizes ($l = 16, 24, 32, 40, 48, 56$) do not change the RQ-RMI size category and only add to the linear scan overhead.

5.2 Training via Approximate Sampling

In NuevoMatch, each neural net in RQ-RMI is trained using supervised learning on a labeled dataset S that is generated in advance. The dataset is sampled from an ordered set of ranges, R, sorted by the ranges' start values. An RQ-RMI model learns the function represented by the ordered set R: it maps an input to the index of the matching range. To learn this function, NuevoMatch samples from it uniformly [26]. This uniform sampling is expensive, as it requires to scan all the ranges and sample from them according to their relative sizes in the function input domain. This sampling must be done for each neural-network (NN) in the RQ-RMI model, sometimes multiple times to achieve the desired accuracy.

Our goal is to modify the sampling process to reduce the number of memory accesses from $O(|R|)$, which can be on the order of tens of thousands per NN, to $O(|S|)$, which is about several thousand per NN. Doing so is not trivial since the training converges faster when the samples are distributed with parameters $(\mu, \sigma) = (0, 1)$.

We make two observations. First, it is possible to analytically estimate the expectation μ and standard deviation σ of a uniform sampling of the NN input domain (see Appendix A.1), and thus enable correct normalization of the samples regardless of the way they are actually sampled. Second, given correct normalization, sampling R in a non-uniform way might only affect the model accuracy but not the lookup correctness, thanks to the search in the rule array (§2.2) that eliminates model approximation errors.

These observations allow us to accelerate the sampling process as follows. We generate a set of 32 samples per batch, each of the form (x, y). First, we uniformly select a range r with index i from R. Second, we uniformly select a value $x' \in r$. We then generate a normalized $x = \frac{x'-\mu}{\sigma}; y = \frac{i}{|R|}$ as in the original algorithm.

In Figure 7b we train a model over a representative rule-set (12-500K, see §7) and get 4-5.3× faster training using approximate sampling.

5.3 Optimized Training Implementation

NuevoMatch uses a hybrid training approach that mixes Python code, TensorFlow, and a custom native library. In contrast, NuevoMatchUP is implemented in C++, which reduces its memory requirements, and takes advantage of the CPU SIMD instructions. Figure 7c shows a 23.8× geometrical mean speedup of NuevoMatchUP over NuevoMatch over all rule-sets. In this experiment we disable all algorithmic optimizations, highlighting the speedup due to the implementation.

5.4 Putting It All Together

Each of the described optimizations in isolation would not suffice to achieve the target performance goals to support the necessary update rate. However, when combined, they allow between two to three orders of magnitude faster training (depending on the rule-set), making NuevoMatchUP suitable for integration with OVS.

6 Implementation

We implement OVS-CCACHE in C as an additional OVS module, and NuevoMatchUP in C++ as an external library (*lib-nuevomatchup*). We add support for OVS-CFLOWS by changing existing components in several OVS modules[1].

Overview. OVS uses *poll mode driver* (PMD) threads for packet processing and *revalidator* threads for integrity. The flows [2] are kept in a dedicated *flow-table*, one per PMD thread, that supports a single writer and multiple concurrent readers. A PMD thread is responsible for inserting new flows into its flow-table, while the revalidator threads remove stale ones.

We modify OVS as follows. We introduce a single *trainer* thread to train all the NuevoMatchUP models used by each PMD thread. In addition, we add *manager* threads, one per PMD thread, for tracking the PMD flows, and create training tasks to accommodate the changes.

Concurrency. We use a fine grained locking with a spinlock per flow-table entry, and limit the number of occurrences in which we modify the flow-table. This mechanism is essential mostly for OVS-CCACHE, in which valid flows frequently migrate between the megaflow cache and the RQ-RMI models.

Training RQ-RMI models. At any given time, there are two instances of RQ-RMI models per *manager* thread: the one that is used by an active classifier in the packet processing pipeline, and the one being trained, referred to as a *shadow* model. In each iteration, a *manager* thread goes over all the flows, checks which are marked for deletion and which are new. Next, it enqueues the request with the modified ruleset to the trainer thread to retrain the shadow model. The rules added during training are updated in the remainder of the active classifier. When the training completes, the active classifier replaces its model with the newly trained shadow model, and the recently learned rules are removed from the remainder. This process repeats whenever the number of flows in the remainder is higher than 10%.

Data-path modifications. The megaflow cache constructs a new hash table whenever it encounters a previously unseen mask, and destroys it when it no longer holds flows. Since in OVS-CCACHE, megaflows frequently migrate between the megaflow cache and the RQ-RMI models, we enable the exis-

---

[1] https://github.com/acsl-technion/ovs-nuevomatchup
[2] In this section we use the OVS terminology and refer to match-action rules of any kind, either megaflows or OpenFlow rules, as flows.

Name	Number of Packets	Unique 5-Tuples	Average Delay Between Packets (μs)
CAIDA-short	100 M	6 M	1.68 ± 69.54
Mawi	237 M	15 M	3.39 ± 9.11
CAIDA-long	401 M	23 M	1.62 ± 119.20

Table 3: Evaluated traces.

tence of empty hash tables to reduce the number of hash table constructions and deletions to bare minimum.

6.1 Updates in OVS-CFLOWS

OVS-CFLOWS offers a new design trade-off for performing OpenFlow rule updates. Specifically, it allows trading the time it takes to activate the updated rules in the data-path for higher throughput during the update. When a network controller updates the rules, it might need to ensure that the updates are installed and visible to the data-path. In OVS and OVS-CCACHE, the acknowledgement to the controller is sent when the rules are installed in the *control-path*. The data-path pulls the rules on demand via upcalls.

In OVS-CFLOWS, we can implement two policies. The *instant update* policy updates the active classifier with the new rules immediately, pushing them into the remainder and thus applying them to the data-path without any delay. The *delayed update* policy first stores the new rules in a temporary structure not visible to the classifier, retrains the shadow model and only then updates the data-path.

As we will see in the evaluation, when a large number of updates is necessary, the instant update policy results in lower throughput while the new rules are being added due to reduced model coverage, but provides lower update latency from the perspective of the network controller. Delayed updates yield higher latency for the controller, but avoid the throughput degradation during the update. On the other hand, with only a handful of updated rules, the immediate update policy achieves low latency without affecting the throughput.

7 Evaluation

We perform end-to-end experiments and provide an in-depth analysis of the system performance using microbenchmarks.

7.1 Methodology

Setup. We use two machines connected back-to-back via Intel X540-AT2 10Gb Ethernet NICs with DPDK-compatible driver. All our tests stress the OVS logic thus the workload is CPU-bound and the network is not saturated.

The system-under-test machine (SUT) runs Ubuntu 18.04, Linux 5.4, OVS 2.13 with DPDK 19.11, on Intel Xeon Sliver

4116 CPU @ 2.1GHz with 32KB L1 cache, 1024KB L2 cache, and 16.5MB LLC. The load-generating machine (LGEN) runs a native DPDK application that generates packets on-the-fly according to a predefined policy, and records the responses from the SUT.

We configure both machines to use DPDK with four 1GB huge pages for maximum performance. We disable hyper-threading and set the CPU governor to maximum performance for stable results.

Synthetic OpenFlow rules. We generate OpenFlow rules using ClassBench [33], the standard benchmark for packet classification [6, 18, 19, 26, 35, 39]. ClassBench creates 5-tuple rule-sets that correspond to the distribution of three applications: Access Control List (ACL), Firewall (FW), and IP Chain (IPC). We generate rule-sets with 1K, 100K, and 500K rules, each size category with 12 rule-sets. We only generate rules for either TCP, UDP, or ICMP IP protocols. The mapping between the generated rule-sets' names to their numbers appears in Appendix A.3.

Traffic traces. The traces are summarized in Table 3 and detailed below.

(1) **CAIDA** [2]. The real trace from the Equinix data-center in Chicago, collected in January 2019. We use CAIDA-short in all experiments except for the one that needs longer trace (Figure 14) where we use CAIDA-long.

(2) **MAWI** [37]. The real trace from a link between Japan and the USA, collected in April 2020.

Adjusting traces to rules. There are no published OpenFlow rules used for processing the packets in the recorded traces. We thus resort to the method used in prior work [26]. Specifically, we modify the packet headers in the trace to match the evaluated ClassBench rule-sets, as follows. For each unique 5-tuple we uniformly select a rule, and modify the packet header to match it. We also set all TCP packets to have a SYN flag. This method preserves the temporal locality of the original trace while consistently covering all the rules.

Packet generation policies. We use minimum-size 64-byte packets to stress the OVS classification logic. We evaluate the system with two load generation methods.
Constant TX rate. To ensure unbiased evaluation, we run the experiments with a constant-rate load generator, and report the highest rate that permits the average drop rate over the whole trace to be below 1%. The first 5% of the packets in each trace are used as a warmup and the associated drops are ignored. We do this as we observe that bootstrapping the megaflow-cache causes many packet drops. With 5% warmup packets, we achieve consistent throughput results.
Adaptive TX rate. We use the timestamps from CAIDA/MAWI packet traces but scale down the inter-packet delay to replay the packets at the highest rate that strives to maintain an average per-second packet drop rate below 1%. To achieve that, we dynamically adjust the
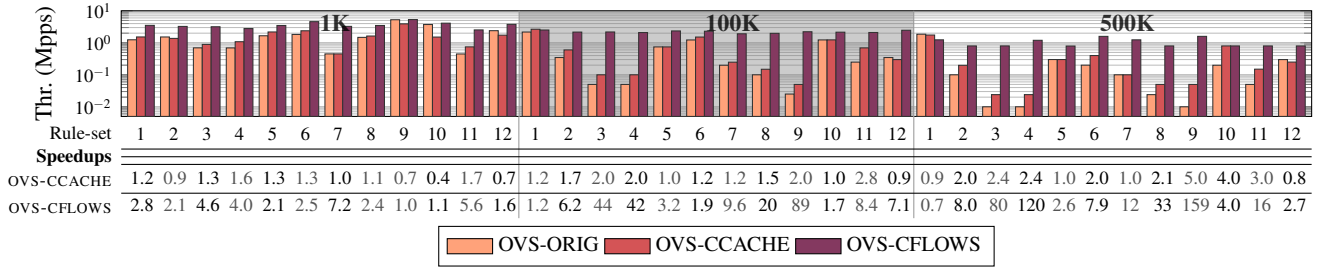
Rule-set	1	2	3	4	5	6	7	8	9	10	11	12	1	2	3	4	5	6	7	8	9	10	11	12	1	2	3	4	5	6	7	8	9	10	11	12
Speedups																																				
OVS-CCACHE	1.2	0.9	1.3	1.6	1.3	1.3	1.0	1.1	0.7	0.4	1.7	0.7	1.2	1.7	2.0	2.0	1.0	1.2	1.2	1.5	2.0	1.0	2.8	0.9	0.9	2.0	2.4	2.4	1.0	2.0	1.0	2.1	5.0	4.0	3.0	0.8
OVS-CFLOWS	2.8	2.1	4.6	4.0	2.1	2.5	7.2	2.4	1.0	1.1	5.6	1.6	1.2	6.2	44	42	3.2	1.9	9.6	20	89	1.7	8.4	7.1	0.7	8.0	80	120	2.6	7.9	12	33	159	4.0	16	2.7

Figure 8: OVS-CFLOWS, OVS-CCACHE and OVS-ORIG on CAIDA-short using constant TX rate. Higher is better.

sending rate once per second: we cut it to half when the drop rate over the last second exceeds 1%, and increase by 50% otherwise. This methods provides a conservative estimate of expected system performance because of its simplistic congestion control which does not aggressively ramp up the throughput after drops.

Measurements. We measure end-to-end performance, i.e., receiving, processing, and sending packets back to the LGEN. We preload all OpenFlow rules into control-path.

OVS configuration. We use the default OVS configuration [22] both for the baseline and our designs: revalidator threads support up to 200K flows, flows with no traffic are removed after 10 seconds, and the *signature-match-cache* (SMC) is disabled. The EMC insertion probability is 20%. Connection tracking is not used. Unless stated otherwise, all experiments use a single NUMA node with one core dedicated to a PMD (poll mode driver) thread and another core dedicated to all other threads. Thus, the baseline OVS, OVS-CCACHE, and OVS-CFLOWS always use *the same number of CPU cores*.

NuevoMatchUP configuration. We use iSets with minimum 45% coverage, and train RQ-RMI neural nets with 4K samples. Similar to [26], we repeat the training until the RQ-RMI maximal error is lower than 128, and stop after 6 unsuccessful ones. We set $l = 40$, namely, each iSet bucket has at most 40 overlapping rules. We use the same RQ-RMI size categories as in Table 1. Due to the use of buckets, the largest size category is never used. We keep OVS's flow matching mechanism that supports an arbitrary number of fields, but limit the iSet construction mechanism to use 5-tuples.

We train RQ-RMI models based on either all megaflows (for OVS-CCACHE) or OpenFlow rules (for OVS-CFLOWS). The model size is determined by the NuevoMatchUP algorithm to allow lowest error, fast training time and low memory footprint.

7.2 End-to-end Performance

Figure 8 shows the throughput comparison of OVS-CFLOWS, OVS-CCACHE and OVS-ORIG (unmodified OVS) for *CAIDA-short* with constant TX rate and without updates to the Open-Flow rule-set. The geometric mean speedups of OVS-CCACHE
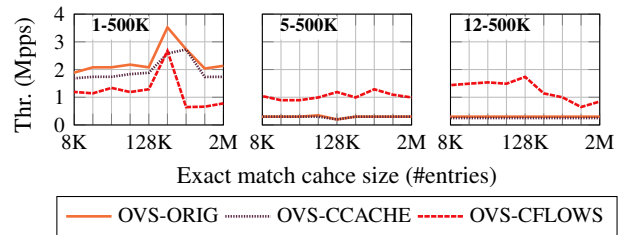

Figure 9: The effect of the exact-match cache size on throughput for the top three fastest rule-sets with 500K rules.

are $1.02\times$, $1.5\times$, and $1.9\times$ for 1K, 100K, and 500K Open-Flow rules respectively. Note that for OVS-CCACHE the computational cache is *constantly updated* with newly installed megaflows.

The same setup with OVS-CFLOWS yields higher speedups. OVS-CFLOWS is $2.6\times$, $8.5\times$, and $12.3\times$ faster than OVS-ORIG for 1K, 100K, and 500K OpenFlow rules, respectively. Not only is OVS-CFLOWS faster than OVS-CCACHE, but it also maintains a relatively stable absolute throughput for 100K and 500K rules. OVS-ORIG performance varies substantially across rule-sets of the same size, whereas OVS-CFLOWS shows more homogeneous behavior. OVS-ORIG has particularly low performance for larger rule-sets (e.g., 3,4 for 500K) due to a massive number of upcalls.

The performance trends with an adaptive TX rate are consistent with those obtained with the constant TX-rate (see Figure 18a in the Appendix). The speedups are still significant but more modest for two reasons: the adaptive TX fails to increase the sending rate fast enough after packet drops, which particularly affects the absolute throughput of faster OVS-CFLOWS. At the same time, it achieves higher average rate for lower-performant OVS-ORIG and OVS-CCACHE because it suffices to slowly increase the rate when the traffic pattern affords that. Rule-set 9-100K and 3-500K are the best illustrations of this effect.

Rule-set 1-500K performs differently from the rest. Here, OVS runs faster with 100K and 500K rules than with 1K rules. We find that this is due to the high temporal locality, which leads to a low upcall rate (over $3\times$ less than in other rule-sets for 500K) and a small megaflow cache. This analysis
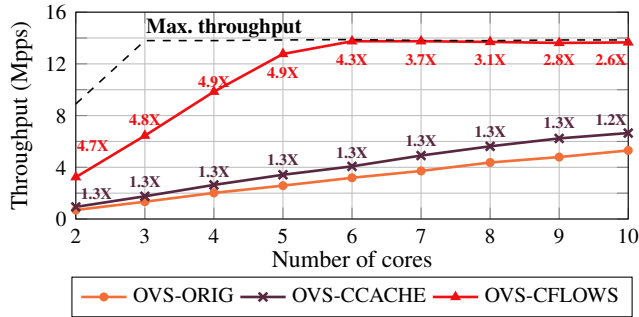
Figure 10: OVS throughput as a function of the number of cores. One core is dedicated to revalidator, manager, and trainer threads. For the rest, we allocate one PMD thread per core. The maximum throughput is measured with only EMC hits. The numbers refer to speedups vs. OVS-ORIG.
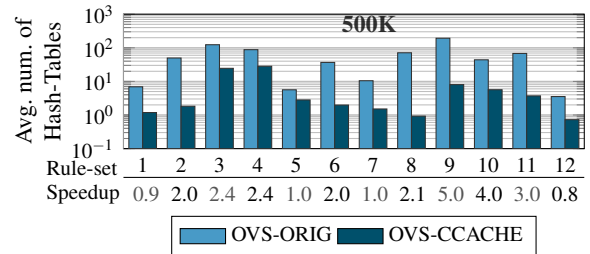

Figure 11: The average number of hash-tables in the megaflow cache on CAIDA-short trace. Lower is better. See full chart in the Appendix (Figure 18b).

is corroborated by the experiments that vary the EMC size (Figure 9). This result motivates dynamic choice between the original and the suggested classification mechanisms as we discuss in §8.

The same experiments on the *Mawi* trace yield low throughput results for OVS-CCACHE and OVS-ORIG using constant TX rates due to the excessive number of drops. For the dynamic TX rate, the geometric mean speedups are: $2.1\times$, $18.2\times$, and $18.7\times$ for 1K, 100K, and 500K rules for OVS-CFLOWS, and $1.02\times$, $1.4\times$, and $1.7\times$ for 1K, 100K, and 500K rules for OVS-CCACHE.

7.3 Sensitivity to OVS parameters

The effect of the EMC size. We take the top three rule-sets with 500K rules that perform best for OVS-ORIG (rule-sets 1,5 and 12), and test their throughput with different Exact Match Cache (EMC) sizes (8K (default) to 2M), see Figure 9. The performance effect of the EMC size depends on the rule-set. The default size (8K) works reasonably well, whereas a too large EMC reduces throughput, likely because of the CPU cache contention. The relative performance of different designs, however, remains largely the same with the EMC of up to 128K entries.

Megaflow cache size. When the OVS megaflow cache reaches its maximum capacity it flushes all its contents. We validated that this never occurs in our experiments. Thus, the megaflow cache can practically grow as necessary, periodically evicting idle (for 10s) flows. This is the most favorable configuration.

Data-path scalability. We add PMD threads and pin them each to a separate core, while dedicating one more core for the revalidator, manager and trainer threads. We use the *CAIDA-short* trace with the constant TX setting, and report the results of a representative rule-set with 1K rules (3-1K) in Figure 10.

This is the best-case scenario for OVS-ORIG because in larger rule-sets it is much slower. We measure the upper bound of the OVS forwarding performance by sending 100M packets that always hit the 8K flows-large EMC (black dashed line). For a 10Gb NIC, the performance saturates at 13.8Mpps, 93% of the line-rate[3].

Figure 10 shows that OVS-CCACHE maintains a constant speedup of $1.3\times$ over OVS-ORIG, even though more PMD threads lead to higher model retraining load. This is because the single trainer thread is fast enough to retrain models from eight PMD threads (nine cores in total on the graph). The additional, ninth PMD thread saturates the trainer. Without training fast enough, the scaling is no longer linear ($1.2\times$ speedup vs. $1.3\times$ for fewer PMD cores). Thus, more PMD threads would require allocating additional trainer cores to maintain the speedup.

OVS-CFLOWS reaches the maximum throughput with five PMD cores (six cores overall), a $4.3\times$ speedup over OVS-ORIG using the same number of cores. OVS-ORIG would have required about 26 cores (linear extrapolation of the current trend) to reach the same performance. Note that in contrast to OVS-CCACHE, models in OVS-CFLOWS are not retrained in the steady state between OpenFlow rule updates, thus the throughput scales linearly with more PMD threads without additional trainer cores.

7.4 Analysis of OVS-CCACHE

Understanding performance variability of OVS-CCACHE. Why does OVS-CCACHE is faster than OVS-ORIG for some rule-sets and is on-par or slower for others? The answer follows from Figure 11 which shows the average number of megaflow cache hash-tables traversed for OVS-ORIG and OVS-CCACHE. Recall that the classification is slower with higher number of hash-tables [3]. The computational cache achieves higher speedups when the number of hash-tables traversed by OVS-ORIG is large enough to justify inference computations instead of memory lookup. As a result, the performance

---

[3]14.88Mpps for 64B packets on a 10Gb NIC, considering bytes of Ethernet preamble and 9.6ns of inter-frame gap.
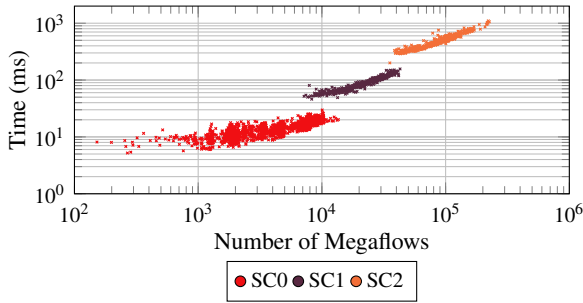
Figure 12: NuevoMatchUP training time in OVS-CCACHE as a function of number of megaflows and the model size category (Table 1). SC0=5, SC1=21, SC2=133 neural nets.
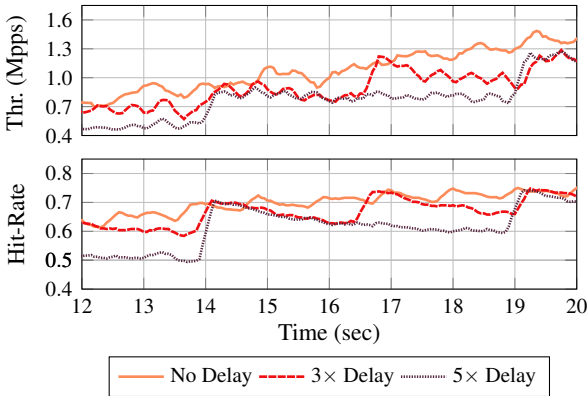

Figure 13: OVS-CCACHE throughput and computational cache hit-rate for different training rates, while adding megaflows. Rapid retraining is critical for high throughput.

savings from using NuevoMatchUP are higher in such cases.

Training in the data-path. In the following experiments, we generate packets at a constant rate of 5 Mpps. This setup saturates the OVS packet-processing pipeline and thus helps highlight the reasons why NuevoMatchUP improves the end-to-end performance.

We measure the actual training time for RQ-RMI models in the data-path during the experiment. To understand the training behavior, we measure the number of megaflows being used and the training time. We show the training time for each of the three used RQ-RMI size categories.

Figure 12 shows that the training time ranges from milliseconds for a small number of megaflows, to about one second for 200K megaflows. For comparison, NuevoMatch reported the training time of 270 seconds for a rule-set with 100K rules which NuevoMatchUP can train in 500ms - 540× faster.

Hit-rate and training time. We further analyze the dynamic throughput behavior of OVS-CCACHE when new megaflows are installed in it by the control path. We use a single rule-set with 100K OpenFlow rules (rule-set 9-100K), and vary the training rate while measuring the throughput.

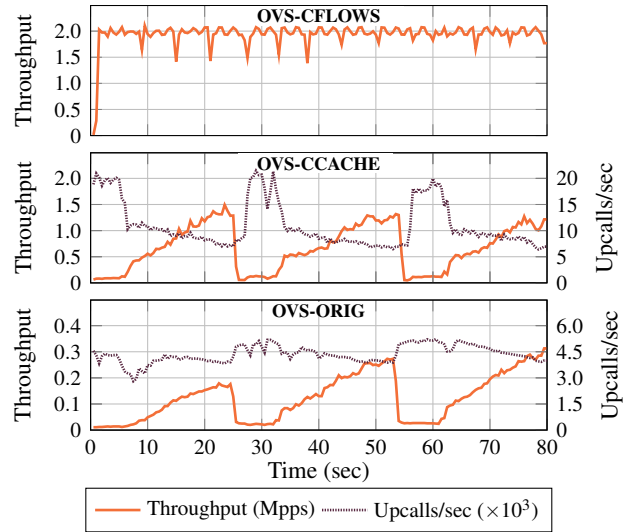Figure 13 shows that when new rules are just added the

Figure 14: The throughput and number of upcalls over time.

throughput decreases initially, but then recovers. This behavior is expected. The rules are first installed in the original megaflow cache, which causes an increase in the number of hash-tables in it and the throughput drops. Also, the hit-rate in RQ-RMI models drops because the new rules are not yet part of the model. However, after the RQ-RMI model is retrained with the new rules, the hit-rate increases back, until the new rules get installed, and so on. Observe that the throughput is lower when the training is slower (i.e., 5× slower than the original rate) since in such cases the system cannot keep up with new rules. This experiment clearly demonstrates the importance of fast training provided by NuevoMatchUP.

Updates in OVS-CCACHE. We measure OVS-CCACHE average update rate for a different number of OpenFlow rules. We see 944, 11.6K and 11.2K updates per second, on average, for 1K, 100K and 500K rules, respectively.

Further inspection reveals that OVS-CCACHE sensitivity to upcalls affect its update rate, similar to the effect presented in Figures 4,5 for OVS-ORIG. Since we cannot explicitly control the upcalls, we test this by artificially delaying NuevoMatchUP updates and measuring the temporal behavior of the throughput, number of upcalls, and iSet coverage. We find that while NuevoMatchUP accelerates the megaflow cache, upcalls are still the dominating factor for its performance. See Appendix A.2 for details.

7.5 Analysis of OVS-CFLOWS

No upcalls in OVS-CFLOWS. We compare the throughput of OVS-ORIG, OVS-CCACHE and OVS-CFLOWS over time, sampled every 500ms. We use the *CAIDA-long* trace, so that each experiment is roughly 80 seconds long, and show the results of a single rule-set with 100K OpenFlow rules (rule-set 9-100K) while keeping the rules unmodified throughout the
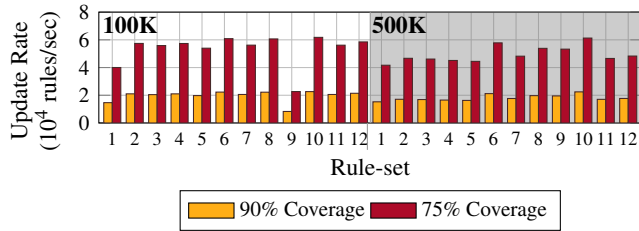
Figure 15: Max OpenFlow rule update rate of OVS-CFLOWS, for maintaining 75% and 90% NuevoMatch coverage.
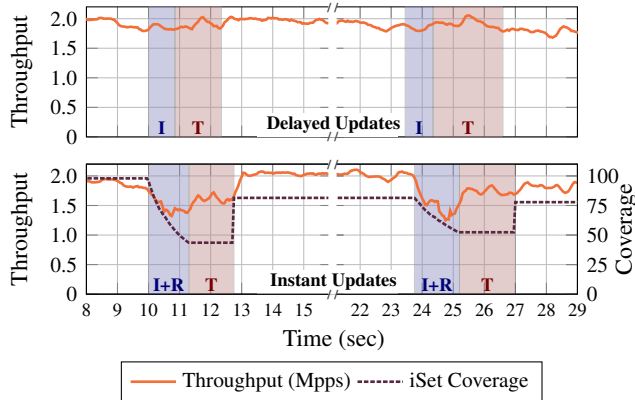

Figure 16: Update policies in OVS-CFLOWS. I: iteration time, R: remainder time, T: training time.

experiments. Other rule-sets behave similarly.

The results in Figure 14 clearly illustrate the benefits of OVS-CFLOWS design (top graph). OVS-ORIG (bottom) and OVS-CCACHE (middle) suffer from significant performance fluctuations directly correlated with the number of upcalls into the control-path. This experiment corroborates our conclusions in Section §3. OVS-CCACHE inherits these performance problems because it simply replaces the megaflow cache with a faster alternative, but uses the same fast-/slow- path split. It does, however, improve the end-to-end throughput. The higher throughput of OVS-CCACHE is the reason why its upcall rate is proportionally higher than in OVS-ORIG.

OVS-CFLOWS avoids the use of upcall mechanism altogether, achieving consistently higher throughput and good scalability for a large number of OpenFlow rules.

OpenFlow updates in OVS-CFLOWS. We estimate the maximum OpenFlow rule update rate in OVS-CFLOWS for 100K and 500K rule-sets, as they pose the main challenge. Unfortunately, we could not measure the maximum update rate experimentally because of the slow OVS control-path that did not allow us to invoke updates back-to-back.

Our estimate of the update rate indicates the number of rules that can be updated per second in order to achieve 75% and 90% coverage by NuevoMatchUP. These are conservative

coverage values that were shown to result in small throughput degradation in NuevoMatch. To estimate, we measure the training time for each rule-set and compute the expected update rate according to the formula in §4.3. The results in Figure 15 show an average of 19K and 51K updates per second for 90% and 75% coverage, respectively. Both size categories achieve similar update rates since the average training time per rule is roughly the same, while the coverage deteriorates slower with more rules. These results assume the use of delayed updates which achieve higher throughput during the update.

Throughput during OpenFlow rule updates. We periodically add bundles of 125K new OpenFlow rules, so the number of rules increases throughout the experiment. We use this number of updates to make the dynamic system behavior over time more visible. We disable the EMC so that the measurements capture only NuevoMatchUP characteristics. We start the experiment with 100K OpenFlow rules, and measure the throughput and iSet coverage over time. We show the results on a representative rule-set (rule-set 9-500K), but the performance is representative of all rule-sets.

Figure 16 compares the delayed and instant update policies (§6.1). For the delayed policy, the time it takes for the data-path to receive the recent changes includes the time to process new rules (iterate over them) and to train, whereas in the instant updates setting, it includes the iteration and remainder update times. The training time depends only on the total number of rules, i.e., 225K and 350K in the first and second training sessions at 10 sec and 24 sec respectively. As expected, the instant update policy causes throughput degradation because the rules are added to the remainder, and thus the model coverage is low. Further, the accesses to the remainder data structure must be synchronized, creating contention. In this case, the use of delayed updates is beneficial as insertions do not cause measurable performance drop.

However, the instant update policy works well when the number of the inserted rules is small. An experiment using bundles of 100, 1K, and 10K new OpenFlow rules yields a 150ms-long drop in throughput with a maximum drop of 2%, 8% and 13% for 100, 1K and 10K rules, respectively. We start OVS with 500 OpenFlow rules and issue an update at $t = 10$ seconds. We use the *CAIDA-short* trace and the constant TX setting with 2.5Mpps. We use the same rule-set as in Figure 16 (rule-set 9-500K); other rule-sets behave similarly. We disable the EMC so the measurements capture only the characteristics of NuevoMatchUP. Figure 17 reports the throughput and iSet coverage within a three second time-frame surrounding the update.

8 Discussion and Future Work

Combining OVS-CCACHE and OVS-CFLOWS. Our evaluation shows that in most cases, OVS-CFLOWS is faster than both
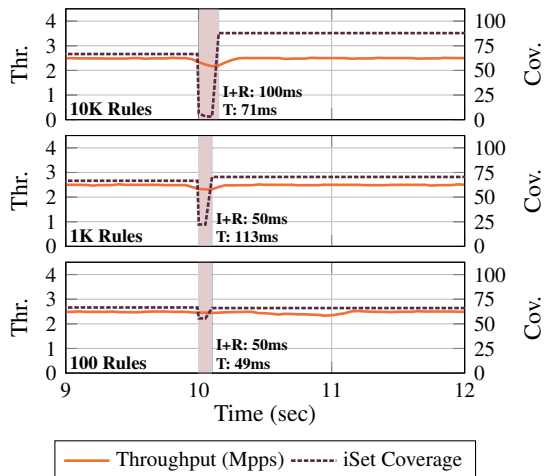
Figure 17: OVS-CFLOWS throughput and iSet coverage upon OpenFlow rule updates using the instant update policy. I: iteration time, R: remainder time, T: training time.

OVS-CCACHE and OVS-ORIG. However, there are cases where it would be desirable to switch between the classification mechanisms dynamically. The computational cache is beneficial when the number of hash-tables in the megaflow cache increases. This can be used to determine when to use it instead of the megaflow cache. Similarly, when the number of control-path upcalls increases, they become the main bottleneck, suggesting the use of OVS-CFLOWS.

NIC OVS offloads. OVS-CCACHE is compatible with the OVS ecosystem, and can be used with in-NIC OVS offloads [20]. In particular, it may accelerate the CPU handling of misses to the hardware OVS cache. Another question is how to use NIC OVS offloads with OVS-CFLOWS. It was shown that NICs become slow when the number of updates gets higher [13]. Thus, switching to OVS-CFLOWS whenever a high number of cache misses is detected may improve performance. We leave it for future work.

In-switch applications. Our work shows a practical use of NuevoMatchUP in packet classification. There are many similar tasks, e.g., longest-prefix matching in switches, which cannot scale due to small on-chip memory. We believe that NuevoMatchUP might help scaling up these tasks by compressing the indexing structure to save on-chip memory.

In-NIC NuevoMatchUP. RQ-RMI inference is a hardware-friendly task. Enabling its execution on the emerging data-parallel accelerators integrated with SmartNICs [21] may improve flexibility of the restricted packet classification offloading logic in NICs today.

P4 OVS. The possibility to use OVS with P4 in addition to OpenFlow was recently suggested [24]. Both the computational cache and computational flows are compatible with P4 as it uses the general structure of match-action tuples which is the fundamental building block for NuevoMatch.

9   Related Work

Packet classification. Software algorithms for packet classification are categorized into decision-tree approaches [9, 10, 18, 19, 28, 35, 39] and hash-table approaches [6, 22, 30]. NuevoMatch [26] is a new approach that shows superior performance for a larger number of rules, hence our choice to use it in this work.

OVS performance. Previous works have highlighted the problem of match-action fragmentation in OVS, and exploited it for mounting denial of service attacks on OVS [3, 4]. Ours is different: it analyses the causes of throughput degeneration and offers a solution.

Machine-learning in the data-path. Several works apply machine-learning models in performance-critical parts of the design, i.e., flash devices [11], RDMA key-value stores [36], programmable switches [38], and NICs [29]. To the best of our knowledge, ours is the first work that applies neural nets and integrates their training into a virtual network switch.

Trading memory accesses for computations. The pioneering work on learned indices [16] and several later works [5, 7, 14, 15, 17, 31] have shown the performance benefits of trading memory accesses for computations using machine-learning models, applying them to data-bases and key-value stores. NuevoMatch [26] extends these concepts and introduces the RQ-RMI data-structure that specializes in range-value queries. Our work improves the training technique of NuevoMatch by several orders of magnitude, making its integration with real-world systems feasible.

10   Conclusion

OVS is a leading virtual networking infrastructure used by many cloud systems. Our work demonstrates two designs which improve its throughput and scalability. We adopt a recent NuevoMatch algorithm for packet classification using neural nets, and integrate it with OVS. Our modifications to NuevoMatch make its use in OVS practical by accelerating its training by over three orders of magnitude. We show significant improvements in both steady-state throughput and update rate for large rule-sets on real-world packet traces. We believe that our work opens new opportunities to practical applications of neural-net based data structures in production networking systems.

11   Acknowledgements

References

[1] The OpenStack authors. The OpenStack project. https://docs.openstack.org/liberty/networking-guide/scenario-classic-ovs.html, 2021.

[2] CAIDA. The CAIDA UCSD anonymized internet traces. http://www.caida.org/data/passive/passive_dataset.xml, 2019.

[3] Levente Csikor, Dinil Mon Divakaran, Min Suk Kang, Attila Körösi, Balázs Sonkoly, Dávid Haja, Dimitrios P. Pezaros, Stefan Schmid, and Gábor Rétvári. Tuple space explosion: A denial-of-service attack against a software packet classifier. In *ACM CoNEXT*, 2019.

[4] Levente Csikor, Vipul Ujawane, and Dinil Mon Divakaran. On the feasibility and enhancement of the tuple space explosion attack against Open vSwitch. *arXiv preprint arXiv:2011.09107*, 2020.

[5] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. From Wisckey to Bourbon: A learned index for log-structured merge trees. In *USENIX OSDI*, 2020.

[6] James Daly, Valerio Bruschi, Leonardo Linguaglossa, Salvatore Pontarelli, Dario Rossi, Jerome Tollet, Eric Torng, and Andrew Yourtchenko. TupleMerge: Fast software packet processing for online packet classification. *IEEE/ACM Transactions on Networking (TON)*, 27(4):1417–1431, 2019.

[7] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David B. Lomet, and Tim Kraska. ALEX: An updatable adaptive learned index. In *ACM SIGMOD*, 2020.

[8] The Linux Foundation. Kubernetes. https://kubernetes.io/docs/concepts/services-networking/network-policies/, 2021.

[9] Pankaj Gupta and Nick McKeown. Packet classification on multiple fields. In *ACM SIGCOMM*, 1999.

[10] Pankaj Gupta and Nick McKeown. Classifying packets with hierarchical intelligent cuttings. *IEEE Micro*, 20(1):34–41, 2000.

[11] Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg Halim, Henry Hoffmann, and Haryadi S Gunawi. Linnos: Predictability on unpredictable flash storage with a light neural network. In *USENIX OSDI*, 2020.

[12] Danny Yuxing Huang, Ken Yocum, and Alex C. Snoeren. High-fidelity switch models for software-defined network emulation. In *ACM HotSDN*, 2013.

[13] Georgios P. Katsikas, Tom Barbette, Marco Chiesa, Dejan Kostic, and Gerald Q. Maguire Jr. What you need to know about (smart) network interface cards. In *PAM*, 2021.

[14] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. RadixSpline: A single-pass learned index. *arXiv preprint arXiv:2004.14541*, 2020.

[15] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Jialin Ding, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. SageDB: A learned database system. In *CIDR*, 2019.

[16] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *ACM SIGMOD*, 2018.

[17] Pengfei Li, Yu Hua, Pengfei Zuo, and Jingnan Jia. A scalable learned index scheme in storage systems. *arXiv preprint arXiv:1905.06256*, 2019.

[18] Wenjun Li, Xianfeng Li, Hui Li, and Gaogang Xie. CutSplit: A decision-tree combining cutting and splitting for scalable packet classification. In *IEEE INFOCOM*, 2018.

[19] Eric Liang, Hang Zhu, Xin Jin, and Ion Stoica. Neural packet classification. In *ACM SIGCOMM*, 2019.

[20] NVIDIA Networking (Mellanox). OVS offload using ASAP² direct. https://docs.mellanox.com/pages/viewpage.action?pageId=39264792, 2020.

[21] NVIDIA. NVIDIA BlueField-2x AI-Powered DPU. https://www.nvidia.com/en-us/networking/products/data-processing-unit/, 2021.

[22] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. The design and implementation of Open vSwitch. In *USENIX NSDI*, 2015.

[23] A Linux Foundation Collaborative Project. Open vSwitch. https://www.openvswitch.org/, 2020.

[24] A Linux Foundation Collaborative Project. Open vSwitch and OVN 2020 fall conference. https://www.openvswitch.org/support/ovscon2020/#D4, 2021.

[25] The DPDK Project. DPDK - data plane development kit. https://www.dpdk.org, 2020.

[26] Alon Rashelbach, Ori Rottenstreich, and Mark Silberstein. A computational approach to packet classification. In *ACM SIGCOMM*, 2020.

[27] Alon Rashelbach, Ori Rottenstreich, and Mark Silberstein. A computational approach to packet classification. *IEEE/ACM Transactions on Networking (TON)*, pages 1–15, 2021.

[28] Sumeet Singh, Florin Baboescu, George Varghese, and Jia Wang. Packet classification using multidimensional cutting. In *ACM SIGCOMM*, 2003.

[29] Giuseppe Siracusano, Salvator Galea, Davide Sanvito, Mohammad Malekzadeh, Hamed Haddadi, Gianni Antichi, and Roberto Bifulco. Running neural networks on the NIC. *arXiv preprint arXiv:2009.02353*, 2020.

[30] Venkatachary Srinivasan, Subhash Suri, and George Varghese. Packet classification using tuple space search. In *ACM SIGCOMM*, 1999.

[31] Chuzhe Tang, Youyun Wang, Zhiyuan Dong, Gansen Hu, Zhaoguo Wang, Minjie Wang, and Haibo Chen. XIndex: A scalable learned index for multicore data storage. In *ACM PPoPP*, 2020.

[32] David E Taylor. Survey and taxonomy of packet classification techniques. *ACM Computing Surveys (CSUR)*, 37(3):238–275, 2005.

[33] David E Taylor and Jonathan S Turner. ClassBench: A packet classification benchmark. *IEEE/ACM transactions on networking (TON)*, 15(3):499–511, 2007.

[34] William Tu, Yi-Hung Wei, Gianni Antichi, and Ben Pfaff. Revisiting the Open vSwitch dataplane ten years later. In *ACM SIGCOMM*, 2021.

[35] Balajee Vamanan, Gwendolyn Voskuilen, and T. N. Vijaykumar. EffiCuts: Optimizing packet classification for memory and throughput. In *ACM SIGCOMM*, 2010.

[36] Xingda Wei, Rong Chen, and Haibo Chen. Fast RDMA-based ordered key-value store using remote learned cache. In *USENIX OSDI*, 2020.

[37] WIDE MAWI WorkingGroup. Measurement and analysis on the wide internet (MAWI). http://mawi.wide.ad.jp/mawi/, 2020.

[38] Zhaoqi Xiong and Noa Zilberman. Do switches dream of machine learning?: Toward in-network classification. In *ACM SIGCOMM HotNets Workshop*, 2019.

[39] Sorrachai Yingchareonthawornchai, James Daly, Alex X Liu, and Eric Torng. A sorted-partitioning approach to fast and scalable dynamic packet classification. *IEEE/ACM Transactions on Networking (TON)*, 26(4):1907–1920, 2018.

A Appendix

A.1 Approximate sampling

We show how to analytically calculate the expectation μ and standard deviation σ of a uniform sampling of an RQ-RMI neural-net input domain. We use the definitions and notations from [26].

RQ-RMI models contain several stages of *submodels* (neural-networks). In each stage, a single submodel is selected based on the output of the previous stage [26]. Let m be an RQ-RMI submodel.

The responsibility R_m of m is defined as the set of all values in $\mathbb{R}$ that might reach m as inputs, formally $I_1 \cup ... \cup I_n$, where $n \geq 1$ and $I_i = [a_i, b_i]$ are sorted non-overlapping intervals in $\mathbb{R}$.

For $1 \leq i \leq n$, define t_i as the sum of all weighted averages of I_j, $1 \leq j \leq i$. For ease of notation, $t_0 = 0$. Note that the intervals $[t_{i-1}, t_i) \subseteq [0,1]$ do not overlap, and their location in $[0,1]$ is relative to the weighted average of I_i.

For all $1 \leq i \leq n$, define the linear function $g_i(z) : [0,1] \to R_m$ as follows:

$$g_i(z) = \frac{b_i - a_i}{t_i - t_{i-1}} \cdot (z - t_{i-1}) + a_i$$

In particular, $g_i(z)$ maps between the weighted average of I_i in $[0,1]$ to $I_i = [a_i, b_i]$. The complete mapping between $[0,1]$ to R_m can be described as the collection of all g_i functions, or as follows:

$$g(z) = \left\{ g_i(z) \big| z \in [t_{i-1}, t_i), 1 \leq i \leq n \right\}$$

Given a uniform random variable $z \sim U[0,1]$, the expectation μ and variance σ^2 of R_m can be described using $g(z)$:

$$\mu = \mathbb{E}[g(z)] \quad \sigma^2 = \mathbb{E}[g(z)^2] - \mathbb{E}[g(z)]^2$$

The two can be manually calculated from the equations above.

A.2 More on updates in OVS-CCACHE

We test the temporal behavior of OVS-CCACHE when facing upcalls and different update rates, similar to the analysis presented for OVS-ORIG (§3). Since we cannot control OVS-CCACHE update rate (§7), we artificially delay adjacent NuevoMatchUP training sessions. We use the same rule-set and trace as in Figure 4, and sample the system's throughput, number of upcalls, and NuevoMatchUP iSet coverage, each 100ms.

The results shown in Figure 19 emphasize the importance of fast updates in OVS-CCACHE, as frequent upcalls cause the iSet coverage to drop to zero after just a few seconds, cutting the throughput by half ($t = 5$ sec). Note that the slow throughput of the system causes it to effectively digest the input at a lower rate, which in turn causes the upcall rate to go down as a result.
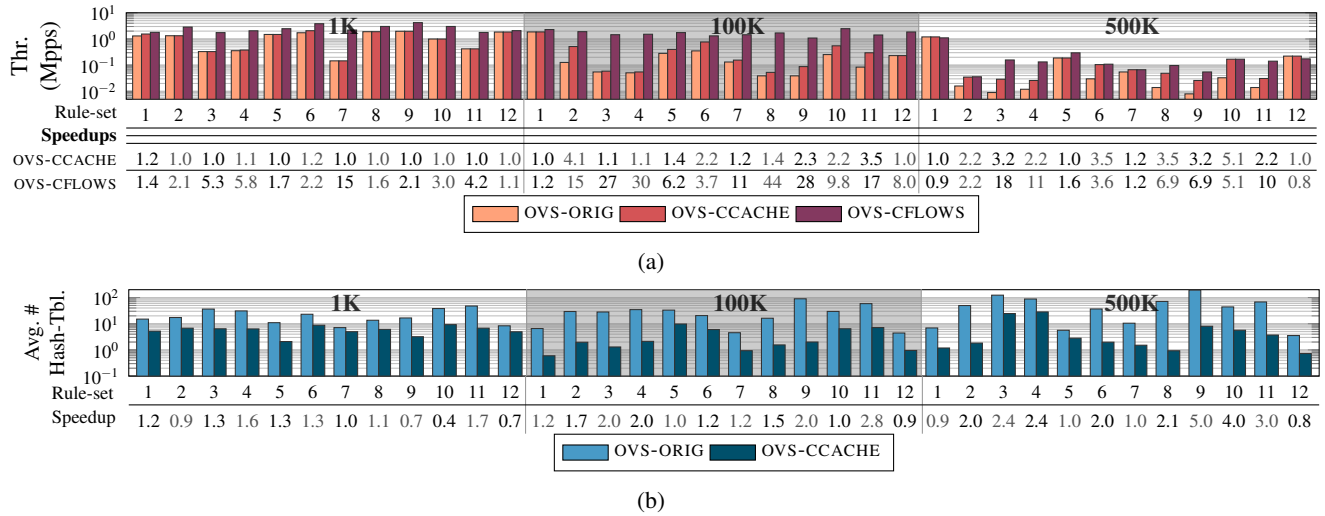
Figure 18: **(a)** OVS-CFLOWS, OVS-CCACHE and OVS-ORIG on CAIDA-short using adaptive TX rate. Higher is better. **(b)** The average number of hash-tables in the megaflow cache on CAIDA-short trace. Lower is better. This is an extended version of Figure 11.
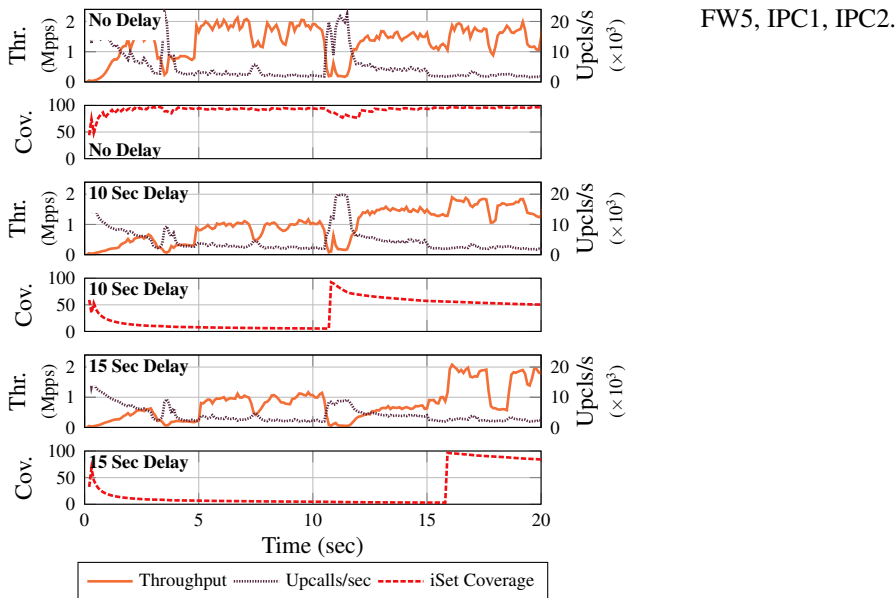

Figure 19: The effect of upcalls and NuevoMatchUP update rate on OVS-CCACHE throughput.

FW5, IPC1, IPC2.

The results also show that upcalls still dominate the throughput as in OVS-ORIG ($t = 11$ sec), thus paving the motivation for OVS-CFLOWS.

A.3 Rule-set names

Rule-set names in Figures 7c, 8, 11, 15, 18a, and 18b by order:
ACL1, ACL2, ACL3, ACL4, ACL5, FW1, FW2, FW3, FW4,

Backdraft: a Lossless Virtual Switch that Prevents the Slow Receiver Problem

Alireza Sanaee[†], Farbod Shahinfar[*], Gianni Antichi[†], Brent E. Stephens[‡]

[†]*Queen Mary University of London, *[*]*Sharif University of Technology,* [‡]*University of Utah*

Abstract

Virtual switches, used for end-host networking, drop packets when the receiving application is not fast enough to consume them. This is called the slow receiver problem, and it is important because packet loss hurts tail communication latency and wastes CPU cycles, resulting in application-level performance degradation. Further, solving this problem is challenging because application throughput is highly variable over short timescales as it depends on workload, memory contention, and OS thread scheduling.

This paper presents Backdraft, a new *lossless virtual switch* that addresses the slow receiver problem by combining three new components: (1) Dynamic Per-Flow Queuing (DPFQ) to prevent HOL blocking and provide on-demand memory usage; (2) Doorbell queues to reduce CPU overheads; (3) A new overlay network to avoid congestion spreading. We implemented Backdraft on top of BESS and conducted experiments with real applications on a 100 Gbps cluster with both DCTCP and Homa, a state-of-the-art congestion control scheme. We show that an application with Backdraft can achieve up to 20x lower tail latency at the 99th percentile.

1 Introduction

Virtual switches (vswitches) play an important role in today's data center networks operation [30, 33, 38, 68]. They are in charge of routing packets to one of the many competing microservices and applications running on a server that are communicating both locally and remotely [48, 66, 86]. They also provide isolation [61, 68, 87], enable load balancing [51], and perform packet encapsulation and decapsulation for secure virtual networking [30, 38, 39].

Virtual switches are fundamentally different from their physical counterpart. A physical switch has fixed port bandwidth, and its draining rate of output queues does not change over time. This is not the case for vswitches, as their draining rate of output queues depends on the ability of connected applications to consume packets. When packets arrive faster than an application can process, queues inside the vswitch fill up and overflow, leading to packet loss. This is called the *slow receiver problem* [21, 44, 60, 73], and it hurts tail network communication latency and wastes CPU cycles, impacting application-level performance [22, 27, 91, 96].

In this paper, we show that slow receivers can manifest at short timescales and cause packet loss even in the presence of state-of-the-art congestion controls such as Homa [72] (§2.1). Moreover, CPU cycles are wasted in handling dropped packets, and this further increases latency and the already high software overheads of current network stacks [21, 72, 73], inflating the problem. Although there are existing approaches to mitigate packet loss (*i.e.*, bandwidth reservation [13, 49, 50], backpressure [31, 43], credit-based hop-by-hop flow control [62], PicNIC [61]), they all have key limitations (§2.2). For example, because virtual ports bandwidth are variable over time, reservation schemes either lead to reduced network throughput or fail to prevent packet loss. Today's backpressure flow control solutions suffer from severe Head-of-Line (HOL) blocking and congestion spreading, leading to reduced throughput across the entire cluster [44, 88, 99] and unacceptable latency for some applications [16, 65]. PicNIC [61, 79], a state-of-the-art solution to provide predictable performance in a multi-tenant data center, incurs high CPU utilization and consequent throughput degradation and HOL blocking for flows sharing a Virtual Machine (VM).

To prevent packet loss from the slow receiver problem, this paper presents Backdraft, a new lossless vswitch. Backdraft prevents packet loss while (1) avoiding HOL blocking, (2) reducing the required CPU cycles, and (3) preventing congestion spreading in the network core (§3). Our main insight is that, unlike physical switches, vswitches have abundant memory that can be used to support a large number of queues.

Leveraging this property, Backdraft assigns a separate queue for every single flow, preventing HOL blocking. To ensure that per-flow queuing is not prohibitive in its memory overheads, we introduce an approach that dynamically reclaims queues from idle flows and resizes them to accommodate in-flight packets from bursty flows.

Also, Backdraft uses separate queues for doorbells (notifications) and packet data to reduce the CPU overhead induced by per-flow queueing that can impact the vswitch performance. In this approach, the vswitch has only to poll the doorbell queue to find where the new to be processed data is located. By keeping the number of doorbell queues low, it is possible to greatly reduce CPU overheads, enabling per-flow data queueing and scaling to 100 Gbps switching performance.

Finally, Backdraft uses an overlay network between communicating vswitches. When a queue inside the vswitch begins to fill because of a slow receiver, Backdraft preemptively sends an Overlay Pause Frame (OPF) to the upstream vswitch responsible for the congestion with pause time and the slow receiver's bandwidth. This is practical because vswitches have a large amount of memory that can be used to store in-flight packets generated by the sender before receiving the OPF notification. Indeed, even buffering a full RTT of packets in a 100 Gbps network, a worst case of 1ms RTT would only require 12.5 MB of space (1 Bandwidth-Delay-Product - BDP), and end-hosts have GBytes of memory.

We implemented Backdraft on top of the BESS vswitch [3, 45] (§4), and evaluated it using a cluster of servers on Cloud-Lab [75] equipped with 10 and 100 Gbps NICs (§6). We experimented with both standard and state-of-the-art congestion controls: in the first case we used unmodified POSIX applications leveraging the TAS TCP acceleration service [56]; In the second, we used Homa [72] with its DPDK implementation. When we ran a distributed application that performs RPCs, Backdraft in conjuction with Homa could lower its tail latency by up to 20x at the 99th percentile. With Memcached, instead, Backdraft could improve its goodput by up to 2.71x when compared to BESS. We also show that Backdraft does not suffer by HOL blocking and because of this can achieve 100 Gbps throughput in a cluster where a slow receiver is present. Finally, we demonstrate that Backdraft ensures high throughput with large number of queues. With 2K queues, throughput is 9x higher than BESS. This paper makes three contributions:

1. We make the case for building a lossless virtual switch by demonstrating the impact of slow receivers on packet loss and network performance using both DCTCP and Homa, a state-of-the-art congestion control algorithm.
2. We introduce Backdraft, a new lossless virtual switch that prevents the slow receiver problem and overcomes the drawbacks of state-of-the-art solutions: It (1) prevents packet loss, (2) removes HOL blocking, (3) increases throughput by eliminating wasted CPU cycles, and (4) avoids congestion spreading in the core network.
3. We implement and evaluate Backdraft on top of BESS using different congestion control mechanisms in a cluster of servers on CloudLab equipped with 10 Gbps and 100 Gbps NICs. We released our code under a flexible
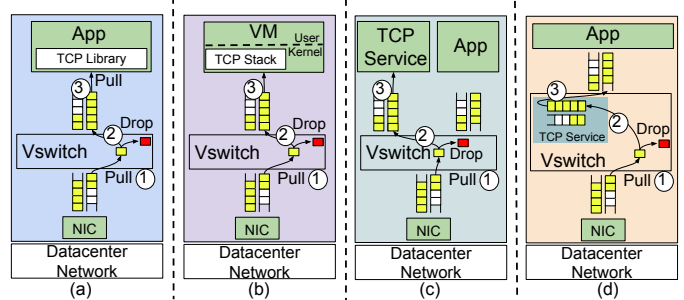

Figure 1: Various deployments of transport layer with respect to vswitches. (a) transport as a library. (b) transport as an OS service. (c) transport as a network function. (d) transport as a vswitch service.

open-source license to enable reproducibility[1].

2 Motivation

Virtual switches use shared memory queues to transmit and receive packets to and from connected end-points (Figure 1). Here, depending on the settings, the transport layer can be directly included into the application as a library (case a) [56], deployed in the kernel of a virtual machine (case b), used as a network service directly attached to the vswitch (case c) [59], or implemented in the vswitch (case d) [68]. Regardless, whenever the vswitch is ready to handle new data coming from the wire, it pulls a packet pointer from one of the NIC queues (point 1), performs processing and places it in the queue associated to the destination endpoint (point 2). Finally, the endpoint pulls the pointer and consumes the data (point 3). If this last step is not fast enough, the queue saturates and packets will be dropped at the vswitch. Notably, the discussed queue is not subjected to transport-level flow control mechanisms, so even if an endpoint has enough memory reserved for incoming packets (for example, TCP's receive window ensures there is space in the receive buffer), it is still possible for packets to arrive faster than the endpoint can process them and eventually get dropped. This issue has been acknowledged in the past, and it is called *the slow receiver problem* [21, 44].

2.1 The Slow Receiver Problem

There are many reasons for slow receivers, including allocation limitations [81], application-level limitations, load imbalance [19, 28, 32, 51, 52, 63, 71, 74], CPU performance variability [17, 25, 37, 42, 54, 66, 82, 97], and CPU/Memory contention [14, 35, 40, 41, 67].

To better understand this, we performed a number of tests on a 100 Gbps cluster (more information available in §6). First, we measured the achievable throughput of data-intensive (i.e., Nginx [8] and Memcached [7]) and network-only applications (iperf3 [6]) using an increasing number

---
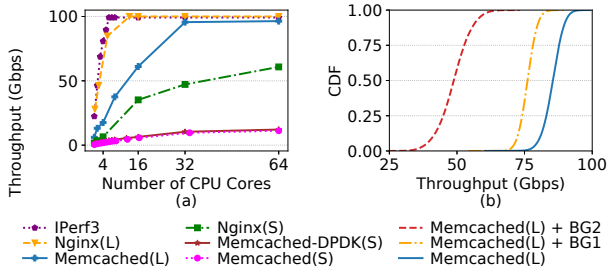[1]https://github.com/Lossless-Virtual-Switching/Backdraft

Figure 2: Maximum achieved throughput by Memcached, Nginx, and Iperf running with DPDK and Linux with large (L) and small (S) response sizes. (a) They require more than 6 cores to achieve 100 Gbps. (b) Memcached exhibits high throughput variability with and without the background workload. (BG1: on isolated cores. BG2: on shared cores)

of assigned processing cores. We also used different packet I/O frameworks (*e.g.*, standard Linux socket and DPDK) and different workloads. For Memcached, we used both small (200 B) and large values (4.8 KB) with sizes inspired by an analysis of caching at Twitter [92]. For Nginx, we served both small (4.8 KB) and large (1 MB) web pages.

Figure 2a shows the result of this experiment. We find that even iperf3, an application that only performs networking functionalities and no other specific processing, cannot hit 100 Gbps throughput with less than 6 cores. For other applications, even 64 cores might not be enough. Further, performance is highly dependent on the specific workload: Memcached using the Linux socket interface and serving 4.8 KB values with 32 cores achieves 16x higher bandwidth than the counterpart serving 200 B values. In contrast, Memcached can achieve 187 KRPS per core when serving 200 B items, while only 78 KRPS when serving 4.8 KB items.

Resource provisioning (OS scheduling) also plays a key role in application behavior [21, 73]. To better understand this, we run Memcached with 32 threads solely on bare-metal servers, where each thread resides on a separate logical core (the number of total logical cores is 64). Then, we use sysbench [58], which only exercises 32 logical cores, along with Memcached on the same machine. We evaluated both scenarios when Memcached and sysbench share CPU cores and when the two applications are isolated on different cores. Figure 2b shows that the Memcached server is unpredictable even *without* a background workload. When it is run with sysbench, its performance degrades by 12% or by 50% depending on the amount of contention. Moreover, the standard deviation of the throughput distribution increases by up to 1.71x.

Even worse, applications behavior can be highly variable and dependent on the workload [92]. We show this with experiments using Memcached and Nginx. To test the former, we used four clients generating a workload resembling the one experienced by Facebook [15]. For the latter, we used sixty single threaded clients requesting data from a copy of
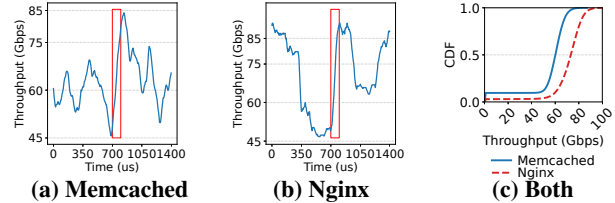

(a) Memcached **(b) Nginx** **(c) Both**

Figure 3: Throughput variability of Memcached (a) and Nginx (b) in a 1.4ms window. Throughput is highly variable over short timespans: *box is 100 µs*. In the box, we can see over 40 Gbps variability in less than 100 µs. (c) CDF of Memcached and Nginx throughput over the entire experiment.
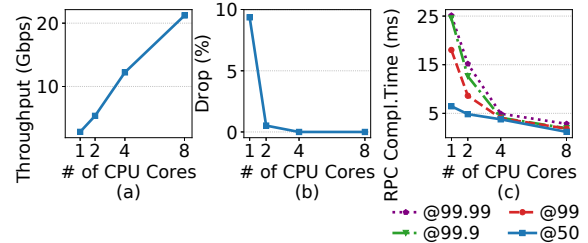

Figure 4: Throughput (a), packet loss (b) and RPC completion time (c) for a bidirectional RPC using Homa, the state-of-the-art transport protocol for data center networks. Packet loss can reach even 10% when only one core is assigned to the server application. RPC completion time can increase by ∼9.3x at 99th.

the NSDI'21 website[2], a fairly light website composed of static pages. In Figures 3a and 3b, we show that performance variability in these applications is temporal. For instance, throughput varies about 45 Gbps in less than 100 µs.

Furthermore, in Figure 3c, we illustrate the CDF of throughput for both Memcached and Nginx. Again, we can see variability: although they can both reach 100 Gbps, but for 50% of the time their throughput stays under 80 Gbps and 60 Gbps for Memcached and Nginx, respectively.

> **Observation I:** Slow receivers are pervasive and can manifest at short timescales.

There are many new congestion control algorithms. However, even new algorithms still suffer from slow receivers. To show this, we ran a number of tests using Homa, a state-of-the-art transport protocol for data center networks [72]. Precisely, we performed a few tests where a client requests Remote Procedure Calls (RPCs) on a server, a dominating pattern in production data centers [57, 86], using a workload similar to the one experienced by Memcached servers at Facebook [72].

In Figures 4a and Figure 4b, we show that when the endpoint cannot process incoming packets fast enough, the drop rate increases. In this experiment, all packet loss occurs at the end-host, and the core network is loss free. This is particularly problematic because packet reception is expensive [69] and CPU cycles spent to eventually drop a packet are wasted resources that can amplify the problem. For example, it has been demonstrated that an increasing loss rate can cause ad-
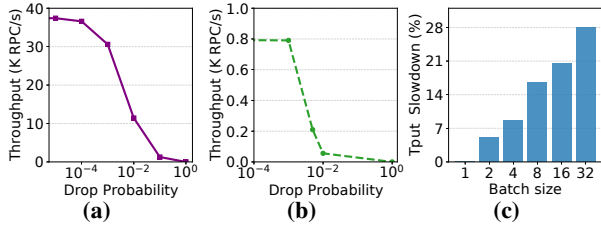
---

[2]https://www.usenix.org/conference/nsdi21

Figure 5: (a) The impact of packet loss on Homa's and, (b) TCP's throughput. Packet loss of 10^{-2} can halve the throughput. (c) The CPU cost of packet admission in PicNIC given different packet batch sizes. PicNIC's out-of-order packet completion queues incur high CPU utilization.

ditional CPU cycles spent in handling the transport protocol, leading to fewer available cycles for data processing [21].

Further, Figure 4c shows that slow receivers lead to an increase in RPC completion time. This is because vswitch queues are shared across flows/RPCs belonging to the same application. As a queue becomes full, flows not responsible for the congestion (victim flows) will experience high latency. Specifically, we can see that even a slight slow down of the receiver application can cause the 99.9^{th} percentile latency to hit values higher than 1ms. Those results show that even small amounts of packet loss can have a dramatic impact on the performance of receiver applications. We also performed a similar set of tests using DCTCP [12] to show DCTCP is also susceptible to high packet loss and report our results in Appendix (§ A.1.1).

To better understand the cost of packet loss, we performed an experiment where the vswitch is configured to drop packets according to a uniform probability distribution. We also used a standard TCP and Homa as transport protocols and measured the maximum sustainable throughput in terms of RPCs-per-second. In Figures 5a and 5b, we can see that even a *small percentages of packet loss* can dramatically impact performance. For example, the maximum sustainable RPCs-per-second can be halved with a packet loss probability of just 10^{-2} when using Homa. Also, the latency of Homa can reach milliseconds scale as packet loss exceeds 10^{-2}, as we show in Appendix (§A.1.2).

> **Observation II:** Slow receivers cause sudden packet loss even in the presence of state-of-the-art congestion control mechanisms. Packet loss impacts network throughput and application service completion time.

2.2 Lossless Vswitching to the Rescue?

Packet loss at the vswitch is the source of many problems. However, there are already a variety of approaches that can be taken to avoid packet loss. These include reservations/rate-limiting, backpressure, credit-based flow control, or a combination thereof. Unfortunately, these have their own key limitations as discussed below and recap in Table 1.

Approach	Prevents packet loss	HOL blocking free	Avoids wasted CPU	Congestion spreading prevention
Rate-limiting [50]	✗	✗	✗	✗
Backpressure [31, 43]	✓	✗	✗	✗
Credit-based [62]	✓	✗	✓	✗
PicNIC [61]	✓	✗*	✗	✗
Backdraft	✓	✓	✓	✓

Table 1: A comparison of existing approaches to reducing packet loss. (*) PicNIC only prevents HOL blocking for flows coming from different VMs.

Reservation Schemes (Rate limiting). One option could be to rate-limit traffic according to bandwidth reservation schemes [13, 49, 50]. Although this is a good option for physical switches, it is not applicable in the virtual context. This is because such schemes assume that the line-rate processing is known in advance and deterministic. While this is the case for hardware switches, it is not for virtual ones.

Backpressure. Another option is to use a backpressure flow control scheme such as PFC [31] or BFC [43]. The main idea here is to send a pause message to the upstream switch before incurring a buffer overflow. Unfortunately, both PFC and BFC have key limitations that prevent them to be used as viable solution in a vswitch. The former might cause HOL blocking [29] and congestion spreading [44, 99] when the PAUSE frame from the vswitch reaches the upstream hardware switch. The second relies on the observation that most flows in a data center network are relatively short at today's 100 Gbps line-rates to avoid HOL blocking from priority hash collisions inside the network core. However, this assumption breaks if slow applications connected to a vswitch are allowed to generate PAUSE messages. In this case, slow receivers will cause congestion spreading, and hash collisions will result in reduced throughput of victim flows from line-rate (100 Gbps) to the rate of the slow receiver.

Credit-based Flow Control. Hop-by-hop credit-based flow control is another mechanism for ensuring zero packet drop [62]. Unfortunately, this technique requires an RTT to request credits and specific support from switches which makes it difficult to be deployed on production networks [24]. Similar to backpressure schemes, credit-based flow control requires packets to be buffered at switches when there are no credits available, leading to HOL blocking.

> **Observation III:** Standard lossless techniques either cannot be used in a virtual context or cause severe HOL blocking and congestion spreading.

Other Approaches (PicNIC). PicNIC [61, 79] is a state-of-the-art solution to provide predictable performance in a multi-tenant data center where per-VM service level objectives (SLO) must be met. PicNIC takes an end-to-end approach to provide backpressure from receivers to senders and aims at preventing HOL blocking at the transmit-side by introducing a packet admission control system where descriptors may be completed out-of-order. This is implemented using a specific

Backdraft Component	Purpose	Expected result
Dynamic per-flow queuing	Avoids HOL blocking, On-demand memory usage	Mitigates tail latency, Improves throughput, Flexible packet scheduling, Prevents pause frame flood.
Doorbell queue	Avoids wasted CPU	Avoids extra pause frame generation, Saves network bandwidth, Alleviates the slow receiver problem.
Virtual switch backpressure overlay network	Avoids packet loss, Vswitch-level flow control, PFC/BFC compatibility	Avoids extra pause frame generation, Saves network bandwidth, Alleviates the slow receiver problem.

Table 2: Backdraft's components and their contributions

feature available in virtio interface [10, 78]. To understand its associated cost, we conducted an experiment where two end-points are connected to a vswitch on the same host. Each end-point is assigned a single core. Then we experimented with both *out-of-order* and *in-order* completion queues in the vswitch. Figure 5c, depicts that the out-of-order packet completion approach is slower than in-order by 20% and 28% when using a batch size of 16 [68] and 32 [3], respectively. Further, this is a baseline with only one core, and these overheads increase with a larger number of cores and queues. Thus, irrespective of application behavior, PicNIC imposes a high toll on performance. Furthermore, while PicNIC can successfully provide predictable performance for flows generated by different VMs, it does not have any mechanisms to ensure isolation between flows coming from the same VM as the out-of-order completion queues have a per-VM granularity, meaning that the slow-receiver problem can still happen and affect all the flows within the same VM.

> **Observation IV:** PicNIC can only isolate slow receivers at a per-VM granularity. It also imposes high CPU utilization and causes throughput degradation.

3 Backdraft Overview

Backdraft is a vswitch that provides lossless networking with higher throughput and lower CPU overheads than lossy switching, and Backdraft does not suffer from HOL blocking or congestion spreading. Backdraft achieves its goals by using three main components: (1) Dynamic Per-flow Queuing (DPFQ); (2) Separate queues for doorbells and data; and a (3) Virtual switch overlay network used for backpressure. Table 2 summarizes the purpose and effect of each component.

Dynamic Per-Flow Queuing (DPFQ): To avoid HOL blocking, Backdraft assigns a separate queue for every single flow in the vswitch, where a flow is an individual TCP connection. However, preallocating queues and memory for the worst case number of flows and burst sizes would be prohibitive. To ensure that per-flow queuing is not prohibitive in its memory overheads, we introduce a new approach that dynamically reclaims queues from idle flows and dynamically resizes queues to accommodate in-flight packets from bursty flows (DPFQ).

By enabling per-flow queueing, Backdraft fundamentally eliminates the HOL blocking caused by slow receivers and
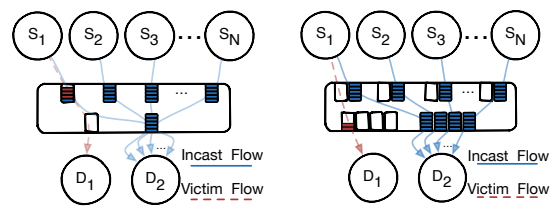
(a) A traffic pattern where using backpressure suffers from HOL blocking.

(b) An illustration of why using separate queues for each virtual switch port avoids HOL blocking.

Figure 6: Queuing and HOL blocking with backpressure.

incasts. HOL blocking only occurs when flows share a queue, and every flow in DPFQ is served by its own queue (Figure 6b versus Figure 6a). DPFQ is possible because end-host memory is not as limited as in physical switches [43, 84]. However, the challenge is ensuring that DPFQ does not incur prohibitive memory overheads even though the number of active flows is potentially large [77]. Over-provisioning leads to memory pressure, while under-provisioning forces flows to fall back to sharing the same queue, potentially incurring HOL blocking.

To solve this issue, Backdraft introduces a new approach to efficiently resize queues on demand. Although all memory for queues and packet buffers is allocated when the process is created to avoid performance stalls, queues are dynamically allocated and reclaimed from flows as they start and stop, and queues are dynamically grown by combining queues as needed to accommodate bursts of packets. This dynamism allows for efficient per-flow queuing without increasing memory overheads. Our insight is that the total amount of congestion that can occur in a vswitch is limited by things like the line-rate of the NIC and not by the number of active flows. Given the same amount of memory, DPFQ enables the same congestion tolerance as a single queue.

DPFQ introduces a new interface to the vswitch. However, it is still possible to support DPFQ without modifying applications. For example, most TCP applications (*e.g.*, POSIX sockets applications) already perform per-flow operations. In this case, only the TCP stack needs to be modified to support DPFQ. Further, Backdraft supports legacy DPDK [47] and Netmap [76] applications that expect a shared queue interface with a vswitch by performing DPFQ inside the vswitch.

Doorbell Queues: The CPU overheads of a vswitch increase linearly with the number of queues that need to be polled [41], and data center workloads may have thousands of flows [18, 77]. Backdraft overcomes this limitation by using separate queues for data and doorbells. For each endpoint, there is a data queue for each flow and a doorbell queue for each core. To send data, an end-point first enqueues packets in data queues then sends a doorbell message to the doorbell queue. This allows the vswitch to poll only an application's doorbell queue to learn about new data.

Doorbell queues also provide a mechanism for applications to communicate scheduling information about the rel-
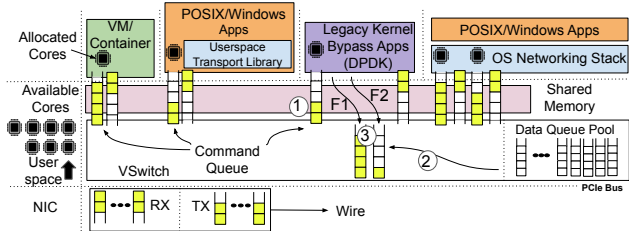
Figure 7: An overview of Backdraft's architecture.

ative priorities and weights of all active flows. Similar to prior work [79, 80, 87], this enables Backdraft to perform programmable scheduling and ensure that the appropriate queues are scheduled first to ensure low latency.

Virtual Switch Backpressure Overlay Network: When combined with backpressure, DPFQ can avoid both packet loss and HOL blocking for traffic local to the vswitch (server). However, if Backdraft runs out of buffer space and data is still incoming from a NIC, it must send a pause frame to the upstream TOR switch connected to the NIC to avoid packet loss when interfacing with a lossless network core, and it must drop packets when interfacing with a lossy network core. Unfortunately, generating pause frames can lead to congestion spreading, while dropping packets has a significant impact on network performance (Figure 5a and Figure 5b).

To avoid such problems, Backdraft builds an overlay network out of vswitches where Backdraft eagerly sends pause messages on the overlay network to the upstream vswitches that are causing congestion and either lazily sends pause messages to the upstream physical switch or lazily drops packets. This enables the local congested vswitch to continue buffering packets while waiting for the remote vswitch to react without causing congestion spreading. Additionally, DPFQ ensures that there is no congestion spreading inside the upstream vswitches because it is possible to pause only the flows responsible for the congestion.

With a lossless network core, the difference between the overlay pause threshold (Th_{over}) and the network pause threshold used for PFC or BFC (Th_{net}) determines the amount of data that can be buffered while waiting for the upstream vswitch to react. If the difference in bytes between these two thresholds is greater than the current network's bandwidth delay product (BDP), *i.e.*, the RTT times the network line rate ($Th_{over} - Th_{net} > RTT * BW$), then it is possible for the overlay network to react to a slow receiver without needing to send a network-level pause message. Because buffering 1ms of packets at 100 Gbps line-rate only requires 12.5 MB of buffering, it is easy to buffer multiple BDPs of packets in a vswitch with low overheads.

4 Design

Applications connect to Backdraft through queues implemented on top of shared memory, and both applications and
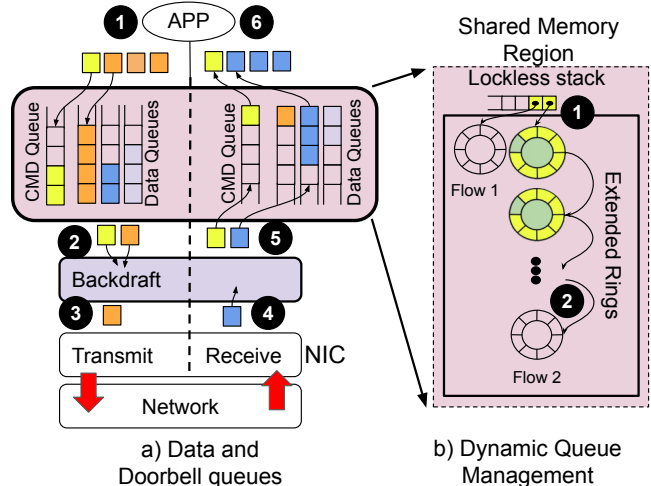

Figure 8: (a) Life cycle of control messages and data messages. (b) Data queue pool memory overview in DPFQ.

the vswitch detect packets by polling. Native Backdraft applications use doorbell queues and data queues in both RX and TX directions. However, Backdraft also supports legacy DPDK applications that only use data queues to send packets as well as standard applications using the kernel networking stack through a custom kernel driver. Currently, Backdraft is designed to be a userspace vswitch although its key ideas are also applicable to kernel-space switching.

Figure 7 provides an overview of Backdraft. First an application sends control messages to the vswitch ①. Upon the arrival of the doorbell message, Backdraft allocates the appropriate data queues ②. Finally, data packets exchange starts ③. Similarly, as new flows start and stop, an application can send more doorbell messages to allocate or release additional data queues. Backdraft supports dynamic queue allocation and resizing via a linked list structure to efficiently manage packet buffers. The rest of this section discusses the design of the Backdraft components in more detail.

4.1 Doorbell Queues

Backdraft uses doorbell queues to reduce the CPU overheads of DPFQ and achieve high throughput. DPFQ increases the number of available queues, and polling them all is inefficient. Checking for outstanding packets costs a memory access, which requires ~100 cycles per queue. There are two ways doorbell queues reduce polling overheads: First, only doorbell queues and not data queues need to be polled. Second, the total number of doorbell queues is kept small. To support parallelism, an application needs at most one hardware thread per doorbell queue.

Figure 8a illustrates the control flow between doorbell and data queues. ① The application generates a doorbell message, notifying the vswitch. ② The vswitch receives outstanding packets. If the destination is a remote server, ③ the vswitch sends the packets to the NIC, and then ④ the packets arrive

at the destination vswitch. Once the packet is at the receiving vswitch, ⑤ the vswitch places the received packet in the appropriate per-flow queue and then generates doorbell message for the application. Finally, ⑥ the application receives a doorbell message and then polls the data. Additionally, all of the command messages in a command queue are read at once in a batch to ensure there is no HOL blocking.

4.2 Dynamic Per-Flow Queuing (DPFQ)

It is important to ensure that DPFQ does not put pressure on memory; hence, DPFQ dynamically reclaims, reassigns, and resizes queues to reduce the memory pressure.

When an application initially connects to Backdraft, the data queue descriptors are negotiated between the vswitch and the application. As applications push packets to buffers, Backdraft allocates individual queues on demand (Figure 8b). To prevent applications' address spaces from being exposed to others, separate shared memory regions and pools of queues are used for each application. This separation of buffering across applications ensures isolation.

Backdraft dynamically resizes queues to absorb packet bursts while minimizing memory overheads. To this end, Backdraft allocates ring buffers of fixed size and then links them together to form and extend queues (Figure 8b-2). Before a ring buffer gets full, Backdraft extends the queue by placing a pointer to a new ring buffer instead of a packet buffer in the overloaded queue. This enables it to learn about an extended queue without any race conditions. Then, once a flow becomes idle, Backdraft reclaims the initial queue into a pool that it can allocate to other queues.

Backdraft pre-allocates all queues at boot time and pushes all the pointers to these queues in a lockless stack. The number of pre-allocated queues can be configured depending on the workload but we used 50 queues for the experiments of this paper. Backdraft benefits from the lockless stack in two ways: First, this structure improves cache efficiency as a pushed pointer can be used immediately from the top of the stack. Second, Backdraft is capable of supporting multiple threads accessing the data queue pool. When a new flow arrives at Backdraft, it borrows a pointer to a queue from the stack and enqueues packet pointers in the queue (Figure 8b-1). If this queue becomes fully occupied, Backdraft borrows another pointer and links it to the previous one as is depicted in Figure 8b-2.

Backdraft does not deallocate empty queues, nor does it leave empty queues allocated to idle flows. Instead, it reclaims empty queues and pushes them back to the lockless stack. This helps Backdraft to reuse reclaimed queues promptly without deallocating them. Backdraft is only responsible for queue assignment/reclamation leading to no race conditions. An entire queue can be reclaimed once there are no outstanding packets in the queue. Full reclamation only happens when a receiver application notifies Backdraft by means of a doorbell message about the emptiness of a data queue. Similarly, for new queues, applications must send a doorbell message to Backdraft requesting a queue corresponding to the new flow.

Both RX queues and TX queues can be extended. RX queues are frequently extended to tolerate bursts. In contrast, TX queues are only extended for flows with large BDPs, and there is no need to extend TX queues beyond a BDP in length. Instead of extending transmit queues beyond a BDP in length, an application can infer that a transmit queue being full is because of congestion or a slow receiver, and DPFQ enables applications to react to congestion. Many applications can simply keep packets buffered inside a TCP stack until the queue drains. However, it is also possible for some applications to mutate or even discard packets to reduce load.

Legacy Interfaces: Backdraft is backward compatible with both POSIX applications and DPDK applications. For the former, there are two ways to interface with Backdraft: (1) Backdraft uses a userspace TCP library that dynamically links to legacy socket applications (TAS [56]). (2) Packets can be received from the kernel through a custom networking driver. This is useful for applications that require features not yet supported by our library, *e.g.*, PF_RING.

4.3 VSwitch Backpressure Overlay Network

When there is congestion because of a slow receiver, Backdraft uses backpressure and sends pauses messages to the upstream sources of traffic to avoid packet losses. However, Backdraft is unique in that there are two different types of pause messages that it can generate: Overlay Pause Frames (OPFs) that are sent on a vswitch-to-vswitch overlay network and network-level pause frames that are sent hop-by-hop across the physical topology by a backpressure flow control scheme like PFC or BFC [43]. Backdraft implements PAUSEs internally by function calls instead of sending PAUSE frames throughout the pipeline because this reduces CPU overheads. PAUSE frames are only created if the PAUSE frame is destined for a remote end-point, which enables Backdraft to provide lossless forwarding across a cluster.

To avoid congestion spreading, Backdraft eagerly generates OPFs. When the occupancy of a receive queue crosses a configurable threshold (Th_{over}), Backdraft generates an OPF and sends it to the upstream Backdraft virtual switch that is causing congestion. Because there is only one flow per receive queue in Backdraft, only one message needs to be generated.

OPFs contain three pieces of information that are used by the upstream vswitch: 1) flow identifier, 2) pause time, and 3) new transmission rate. When an upstream vswitch receives an OPF, it pauses the input queue for the specified pause time, and then it applies a transmission rate-limit on the input queue.

Although prior backpressure schemes only send a pause time, sending a rate in an OPF is important to avoid persistent on/off congestion bursts from transmitters restarting after

being paused. To support this, Backdraft tracks an estimated receive rate (R_{recv}) using an exponential weighted moving average (EWMA) for each receive queue as it delivers packets, and it uses this rate when generating an OPF. The pause time is set as $(Th_{curr} - Th_{goal})/R_{recv}$ where Th_{curr} is the current length of the queue and Th_{goal} is the target queue length, which is equal to the batch size of packets read by the TCP stack by default to help ensure efficient CPU utilization.

The biggest concerns with respect to choosing values for Th_{goal} and Th_{over} are in avoiding starvation and reducing CPU overheads. Starvation is possible if the receiver vswitch either underestimates the end-point's rate or sets too large of a pause value. Th_{goal} provides headroom to avoid this, and if starvation is observed to be a problem with a running application, both the application and the vswitch can increase this value. In contrast, to avoid congestion spreading, it is desirable to set as large of a value for Th_{net} as possible because Backdraft generates PFC/BFC messages that will be processed by the upstream switch when this threshold is exceeded. This value can be as large as the maximum length of the queue minus the 1-hop bandwidth-delay product between the server and its TOR switch (1-hop RTT $\times$ line-rate).

On the whole, sending OPF messages significantly reduces CPU utilization by preventing packet drops. However, to reduce the CPU overheads of OPF messages, Th_{over} is set to be at least one batch size of packets larger than Th_{goal} to not interfere with batching. Further, to avoid excessive OPF generation, Backdraft generates a new OPF message only if the previous OPF message pause time has gone past. When the pause time passes, Backdraft checks the queue length to decide whether to generate another OPF message or not.

5 Implementation

Backdraft builds upon the BESS virtual switch [3] (commit 0145a1c). We have extended the TAS TCP stack [56] (commit a1c158f) to support TCP legacy applications. Further, we have implemented a Homa open-loop app based on the Homa DPDK library (commit 392b577) and altered the DPDK driver to interface with Backdraft. Our changes to BESS amount to about 3.5K LOC, and our changes to TAS and Homa required about 100 and 500 LOC, respectively. Apps running TAS and Homa both connect to BESS via a DPDK vHost user port.

6 Evaluation

In this section, we evaluate the performance of Backdraft and demonstrate that Backdraft is able to prevent packet loss while providing 100 Gbps switching capabilities and without incurring in HOL blocking.

Experimental cluster: We used two different types of clusters from CloudLab [75]. On the first, we were able to use PFC to perform experiments with a lossless fabric. This clus-
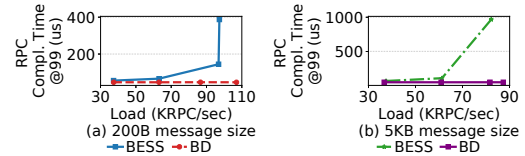

Figure 9: Performance of a victim RPC with Homa in the presence of an increasing load generated by a competing application. We used two different message sizes and either BESS or Backdraft as vswitch. The victim RPC is less impacted by the competing workload in the presence of Backdraft.

ter has 6 servers, and each server has an Intel Xeon E5-2640 CPU running at 2.40 GHz with 64 GB of RAM and a 10G ConnectX4-L NIC. These servers are connected via a Mellanox SN240 10 Gbps TOR switch. We used a second cluster with 4 servers to perform experiments at 100 Gbps. Each server has an AMD EPYC 7452 64-Core CPU running at 2.30 GHz with 128 GB of RAM and a 100 Gbps ConnectX-5 NIC. These servers are connected via a Dell Z9264F-ON switch.

Applications: When experimenting with TCP, we leveraged the TAS TCP acceleration service to connect three unmodified POSIX applications to Backdraft: Memcached [7], Mutilate [64], and a custom distributed application that performs RPCs. To perform experiments with Homa, we utilized the Homa DPDK implementation [4], which unfortunately does not have any native support for applications. We overcome this problem by developing an open-loop RPC application on top of Homa. Because PicNIC [61] is proprietary software, a head-to-head comparison is not feasible.

Performance metrics and comparison points: Our experiments focus on four main metrics: packet drop rate, CPU utilization, throughput, and 99th percentile request completion time latency. We also compared Backdraft against two variations of BESS virtual switch: lossy (default), and a lossless variation which generates PFC messages.

Key results: With Backdraft, the Homa-based RPC application achieves 20x lower tail latency at the 99th percentile (§6.1). Further, Memcached achieves 1.9x higher goodput with Backdraft (§6.2). In a lossless multi-node scenario, Backdraft prevents congestion spreading in the network core (§6.3). In a 100 Gbps setup, Backdraft avoids HOL blocking and reaches 100 Gbps even in presence of slow receivers (§6.4). Finally, Backdraft supports 16 K queues without any throughput slow down (§6.5).

6.1 Backdraft Complements Homa

Our first experiment demonstrates that Backdraft complements Homa. In this experiment, we used two different machines in the 100 Gbps cluster: one of them hosting three client applications and the other two server applications. Each client/server application is assigned to a single CPU core. We used two clients to generate fixed-size RPC requests towards one server. The other client, instead, generates requests to-
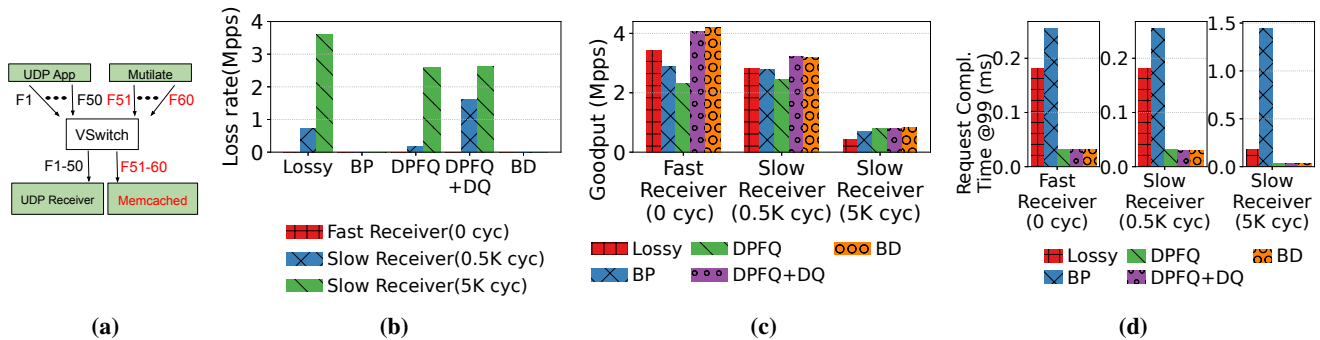
Figure 10: Performance of the individual components of Backdraft in presence of slow receivers when handling a Memcached TCP incast (10 flows) workload with a background UDP workload (50 flows). (a) Experimental setup (b) Aggregate drop rate, when the UDP server spends on average 0/0.5 K/5 K extra cycles on every delivered packet. Slower receivers have more detrimental impact on the performance. (c,d) detailed breakdown of goodput and latency impact of Backdraft. Backdraft improves tail latency up to 5.65x compared to BESS, and 45.2x compared to BESS augmented with PFC at the 99th percentile while achieving 1.9x higher goodput.

wards the remaining server using the Facebook Memcached workload [15].

We compare the RPC performance of the client using the Memcached workload when using either BESS or Backdraft as vswitch. In Figure 9a and Figure 9b, we show the results when fixed-size RPCs are 200 B and 5 KB, respectively. When the RPC load increases, the completion time with Backdraft remains stable, while it inflates by over 20x with BESS. The poor results experienced with BESS are a consequence of its single queue design. In contrast, Backdraft keeps tail latency low because each `RPC_ID` occupies a single queue in the vswitch. This way, Backdraft removes HOL blocking of various RPCs with different service times.

Homa and Backdraft have strong synergy. Homa eagerly sends `RESEND` control messages to peers (`RESEND_INTERVAL` = 2 μs [5]). This enables Homa to detect packet loss proactively, resulting in better tail latency. The CPU overhead of this task can be prohibitively high in presence of packet loss. For instance, without Backdraft, the CPU usage of Homa increases 8-10% because there are more outstanding messages to manage due to loss. Backdraft avoids wasting CPU cycles by preventing packet loss, enabling the transport protocol to provide better performance.

6.2 Per-Component Analysis

To provide a performance breakdown of the benefits of the different Backdraft'components, we created a scenario in a single host where background UDP packets destined to a slow receiver (50 flows) compete against a Memcached application with 10 active flows generated by Mutilate (Figure 10a). Here, we considered three cases: (1) the receiver spends 0 cycles processing the received packet; (2) the receiver spends 500 cycles; and (3) the receiver spends 5000 cycles. For context, Facebook's Katran load balancer spends 100 cycles per packet [20], and complex functions like range queries in key-value stores can easily take more than 1 K cycles.

Figure 10 shows the results of this experiment. With *Lossy*, we consider the default behavior of BESS, while

BP is BESS with PFC enabled. DPFQ, DPFQ+DQ, and DPFQ+DQ+ON (BD) show the incremental benefits of different Backdraft components: dynamic per-flow input queueing (DPFQ), doorbell queues (DQ), and the overlay network (ON). BD indicates our final system with all components.

Figure 10b shows packet loss rates given the slow receiver application (UDP receiver) in Figure 10a. BESS with PFC and Backdraft both report zero packet loss. Without PFC for BESS and without the overlay network for Backdraft, packets may be dropped. Packet loss occurs in both the slow and fast flows, and it is more problematic in the presence of a slow receiver. DPFQ+DQ reduces CPU overheads and can forward at higher throughputs than just DPFQ. This results in even more packet loss at the receiver. This packet loss, however, is avoided by introducing the overlay network (ON). Backdraft prevents packet loss and achieves higher throughput and lower tail latency.

Next, Figure 10c shows the aggregate goodput achieved by the applications (UDP and Memcached). Backdraft always outperforms BESS, even when the latter is augmented with lossless capabilities using PFC. In this experiment, for the 0, 500, and 5 K cycle receiver, Backdraft achieves 22%, 10%, and 200% and higher goodput than the lossy counterpart, respectively. Backdraft also mitigates tail latency at the 99th percentile by up to 5.65x.

Looking at the individual components, we find that DPFQ has a negative impact on performance because polling more queues consumes more CPU cycles. However, combining DPFQ and doorbell queues (DPFQ+DQ) improves goodput by reducing cycles spent polling. This effect is more visible in the presence of a fast receiver, as the faster the receiver the more packets need to be process by the vswitch. The last component (ON) enables Backdraft to prevent packet loss. This is illustrated in Figure 10b.

Figure 10d shows the latency experienced by Memcached. In this figure, Backdraft similarly outperforms both BESS configurations. The BP bar shows that naively applying a backpressure mechanism dramatically increases network latency, and this is mainly an effect of HOL blocking. DPFQ, in
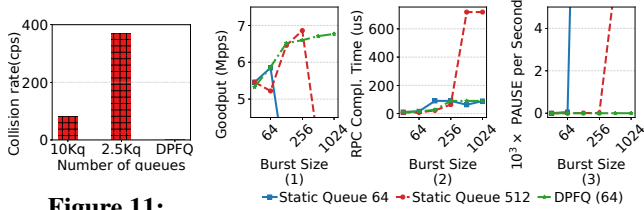
Figure 11: Compares collision rate of a skewed workload when varying the number of queues.
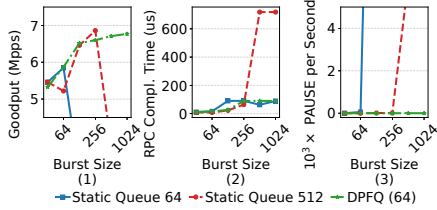

Figure 12: Shows goodput, latency and PAUSE rate of short queues, long queues verses DPFQ. Backdraft can absorb different burst sizes compared to static queue allocation with PFC.
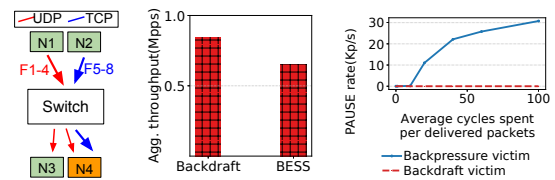

Figure 13: Performance of Backdraft overlay network in a cluster-wide experiment. Backdraft achieves higher throughput and avoids extra PAUSE frame generation in presence of a slow receiver. (a) Experiment setup. (b) Overall throughput. (c) Pause frame generation rate due to a slow receiver.

Config	Tput (Gbps)	Pause (Kfps)	Drop (Mpps)
BESS Lossy + Lossy Network	(2.36,21.85)	N/A	(1.6,0)
BESS Lossless + Lossy Network	(2.66,19.29)	(2.8,0)	(1.3,0)
ON + Lossy Network	(2.3,21.98)	(0,0)	(0,0)

Table 3: Virtual overlay network performance results (Victim,Non-victim flow)

contrast, keeps the overall latency low, even in the presence of a slow receiver. This is because providing per-flow queues prevents HOL blocking.

To better understand Dynamic Queue Allocation (DQA), we used a sample client application generating approximately 100 K flows to a server sink application on the same machine where only 1 K flows are active at any point in time. We compared two different policies for queue allocation: a static number defined at configuration time and a dynamic. The former assigns flows to queues using an RSS (Receive Side Scaling) hash function, the latter creates a new queue anytime a new flow shows up. Figure 11 shows that when using only 2.5 K queues, the collision rate is high, even if only 1 K flows are active. Having 10x more static queues than active flows helps, but still collisions occur. In contrast, DPFQ avoids wasted memory and achieves a zero-collision rate thanks to its per-flow queueing mechanism. Each ring buffer consumes about 20 B. 10 K queues will consume 200 KB, where DPFQ allocates only 1 K queues since we have 1 K active flows, requiring only 20 KB. This is a 10x reduction in memory utilization in addition to the reduction in collisions.

Next, we evaluated Backdraft's ability to absorb packet bursts by extending queues by performing an experiment where a sender pushes different batch sizes (64 to 1024) to a receiver. The receiver is attached to a vswitch on the same server and pulls packets in large batches of 1024. This experiment compares Backdraft against two different configurations of BESS augmented with PFC: one with short queues, the other with long. Short queues are more likely to generate PAUSE frames at a higher rate, whereas longer queues are less likely.

Figure 12 shows that, when increasing the burst size, lossless BESS with short queues causes a high PFC PAUSE frame generation rate that would hurt application performance in terms of goodput and tail latency. Although long queues reduce the PAUSE frame generation problem, this is at the cost of increased latency. In contrast, this experiment shows that Backdraft is able to absorb variations, particularly in a bursty workload with its dynamic queue extensions. It is the only configuration that does not generate PAUSE frames. Moreover, Backdraft maintains high goodput with DPFQ because the cost of queue extension is relatively low.

Finally, we also measured the overheads of extending queues in DPFQ and found that it is small. The number of cycles required to extend queues fluctuates between 24 and 350 cycles, and this value is dependent on caching. This shows that the overheads of DPFQ are low, especially when amortized over all of the packets in the added queue, which has a default size of 64 packets. Further, if desired, Backdraft's queue size can be configured as a parameter based on the measured overhead according to the user's preference for performance versus memory efficiency.

6.3 Multi-node Performance

This section studies the behavior of the overlay network between vswitches used in Backdraft. To do this, we used a cluster of four different servers. Each server is running its own vswitch, and they are connected through a physical switch with PFC enabled. We generated background UDP flows competing with TCP victim flows (Figure 13a) and compared the results when using either BESS or Backdraft as a vswitch. Backdraft achieves higher aggregate throughput than BESS (Figure 13b). This is because Backdraft sends PAUSE frames through the overlay networks as soon as it notices queue buildup. This is not done by BESS, which in turn induces the physical switch to send PFC PAUSE frames and trigger congestion spreading inside the network.

Figure 13c shows the number of PFC PAUSE frames sent by the receiving server to the upstream PFC enabled switch as a receiver gets slower. In this scenario, BESS causes the physical switch to also generate PAUSE frames. However, this does not happen with Backdraft because the overlay network pauses the flow for the slow receiver before queues fill up.

We also compared the performance of BESS and Backdraft using two nodes in the 100 Gbps CloudLab testbed connected
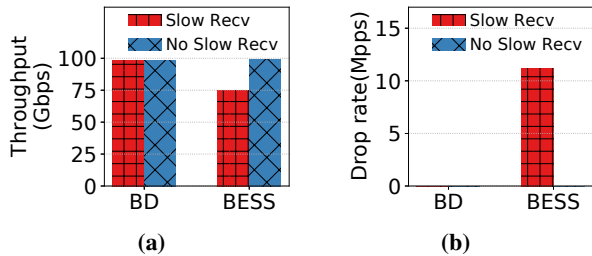
Figure 14: Multithreading in Backdraft, (a) Aggregate throughput, (b) Drop rate of victim flow. Backdraft achieves 100 Gbps using multiple cores while ensuring zero drop at the vswitch. In this experiment, applications are allocated enough number of cores to drive 100 Gbps.
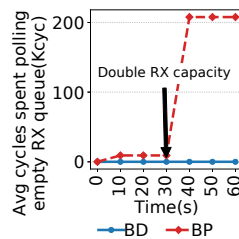
Figure 15: Backdraft has no starvation. Backdraft adjusts the sender rate using overlay messages to ensure enough buffering at the receiver's queues.
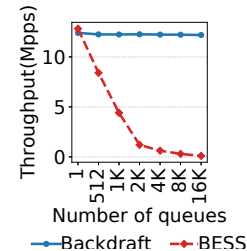
Figure 16: Backdraft, compared to BESS, sustains high throughput with a single core while managing large number of queues as it mitigates the polling overhead.

by a lossy switch. Here, we used one server to send two UDP flows towards another machine where one receiver is slow and the other instead is a victim. Table 3 reports our results. When using standard BESS (lossy) with a lossless network core (first row), the overall throughput is high. However, it also suffers from a high degree of packet loss. When, instead, using BESS generating PFC frames (lossless), the throughput is reduced and a considerable amount of packet loss still appear, as the network core is lossy. Finally, due to overlay messages, Backdraft it is able to avoid packet losses, while keeping network throughput high.

Finally, we performed an experiment to demonstrate that the overlay network in Backdraft does not suffer from starvation, even when the rate of the slow receiver is variable over the time. In this experiment, one machine is sending packets towards a slow receiver. Initially, the destination polls packets at rate 3 Mpps, then it doubles its rate at time T_{30}. In Figure 15, BP (BESS with PFC) suffers from starvation and the receiver spends its extra cycles polling instead of processing packets. In contrast, at $T = T_{30}$, Backdraft detects a change in the receivers rate and increases Th_{goal} to avoid starvation for the rest of the application's life.

6.4 100 Gbps Forwarding Performance

To show that Backdraft can achieve 100 Gbps throughput regardless of the presence of slow receivers, we performed an experiment where an 8-core sender is generating a heavily skewed workload consisting of 12 flows (11 fast flows and 1 slow flow) towards an 8-core receiver. To cause a slow flow, one of 8 cores of the receiver application is slowed down in this experiment. When using BESS, the slow flow will eventually block the others, forcing the vswitch to drop packets due to a lack of queue descriptors at the receiver's RX queues. In contrast, Backdraft does not suffer from this problem because of its ability to dynamically resize queues and send overlay PAUSE frames.

Figure 14 shows the aggregate throughput for all flows and the drop rate for the victim flow in presence of slow receivers in this experiment. With BESS, this experiment results in high packet loss and a decreased throughput of ∼75 Gbps

even though there is only one core receiving slower than the expected pace. In contrast, Backdraft achieves full line-rate without any packet drops. Backdraft sends overlay messages on a per-queue basis to notify the upstream sender to reduce its rates. This allows Backdraft to utilize the extra bandwidth for the other flows in order to drive the 100 Gbps line-rate.

6.5 Backdraft Scalability

Finally, we assessed the scalability of Backdraft in terms of its throughput and memory requirements.

The impact of number of queues on performance. To demonstrate the benefits of doorbell queues, we performed an experiment where an application sends packets from UDP flows in a skewed pattern based on the Zipfian distribution, and we compared the throughput achieved between doorbell queues (Backdraft) and polling every queue (BESS).

Figure 16 shows the aggregate vswitch throughput when a single core is allocated to the switch as we vary the number queues. With a small number of queues, both Backdraft and BESS perform similarly, which shows that the overheads of doorbell queues are quite low. However, when the number of queues increase, only Backdraft maintains its throughput.

The amount of memory needed varying network RTT. Finally, we performed an analysis of the memory overheads of Backdraft to demonstrate that this is not prohibitive. In order to avoid congestion spreading, Th_{net} must be sufficiently larger when compared to Th_{over} so that packets can be buffered during the time it takes for the source of the congestion to pause and adjust its rate. The increased memory overheads of Backdraft are small and can be estimated by bandwidth-delay product for different network line-rates. For example, a 100 Gbps network with a 1ms RTT only requires 12.5 MB of buffering to avoid congestion spreading. Further, it is important to note that DPFQ ensures that this buffering requirement is for the entire switch and not per-flow.

7 Discussion

Slow NICs. NICs may be slow and unable to achieve line-rate, and this can cause packet loss [44, 83]. If a slow NIC

participates in the overlay network by generating OPFs, it can avoid both packet loss and congestion spreading.

Slow virtual switches. There are many reasons a vswitch may be slow, including CPU limitations, memory bandwidth limitations, and insufficient LLC cache [34, 36, 89, 95], and packets can be dropped at the NIC when a vswitch cannot keep up with the ingress rate, resulting in a slow vswitch problem. This can be solved by offloading part of Backdraft's processing onto a programmable NIC [2, 9, 39, 46, 87]. This should be feasible because recent developments in NIC designs have brought models that provide a large amount of on-NIC memory that can be used for Backdraft. For example, Xilinx Alveo NICs support High Bandwidth Memory (HBM), fast memory that is directly embedded on-chip in an FPGA [11, 53].

RDMA support. Backdraft can support 2-sided RDMA verbs by monitoring the length of receive queues in the application or a library and generating OPF messages to transmitters as appropriate. Further, if offloaded onto a NIC, Backdraft can mitigate the effect of the slow NIC problem for 1-sided verbs and complement the sender-based approaches that can be used for congestion control [55, 68, 98].

Programmable packet scheduling. If Backdraft is deployed without enough memory, multiple flows have to share the same queue. Although this can cause HOL blocking, this can be mitigated with opportunistic packet scheduling. For example, Backdraft could employ software solutions like Eiffel [80] or hardware ones like AIFO [94] and PIFO [85] if Backdraft is offloaded to a programmable NIC.

Linux kernel compatibility. Backdraft is implemented using DPDK. Thus, all of the traffic coming from the NIC bypasses the Linux kernel. However, we believe that the same design principles are applicable to the Linux kernel. Further, being implemented in userspace does not even preclude Backdraft from interfacing with the Linux kernel networking stack. For example, Backdraft can use a custom kernel driver to interface with traditional applications, and the recently proposed `AF_XDP Poll Mode` driver [1] enables DPDK applications to natively support the AF_XDP socket and retain compatibility with the Linux tools that operators expect [90].

8   Related Work

Slow receiver problem. Past research has acknowledged the slow receiver problem in the context of the overheads of the Linux networking stack [21], Linux-based transport protocol implementations [73], and production networks from Microsoft [44], and Google Swift [60].

Virtual switching. Snap, Andromeda, and PicNIC all perform lossy vswitching [30, 61, 68], which drops packets. On the other hand, Zfabric, NFVNice, and zOVN are lossless vswitches. These, however, suffer from HOL blocking as they share queues among active flows in the vswitch [26, 27, 59]. Moreover, unlike Backdraft, none of these approaches address the slow receiver problem. Similarly, FreeFlow ensures high performance by using shared memory, but it does not consider packet loss problem at the end-hosts [93].

Packet scheduling and rate limiting. Backdraft is compatible with Eiffel and Carousel and can mitigate their CPU utilization overheads with its command queue [79, 80]. Similarly, hyperplane can be used to reduce the CPU polling overheads of Backdraft [70]. EyeQ is a related system that builds an overlay network that performs rate-limiting [50]. However, EyeQ pays high CPU utilization overhead when rate limiting, and EyeQ works at millisecond-scale, which is not fast enough to address the slow receiver problem.

Congestion control. In addition to Homa, there are other important new congestion control algorithms like Google's Swift, which performs fine grain time stamping to identify the congestion source (end-host, network) [60]. Similar to how we have found that Backdraft is complementary to Homa, we expect that Backdraft is complementary to Swift as well.

Flow control. Backdraft is complementary to flow control protocols designed to provide a lossless core network. For example, Backdraft is complementary to PFC because it strives to minimize the PAUSE frames sent across the network. PCN ensures high throughput for victim flows if congestion spreading occurs and is also complementary to Backdraft [23]. BFC is a new backpressure flow control protocol intended to replace PFC [43]. Backdraft solves a key problem that arises with deploying BFC in practice. This is because BFC assumes that flows can be received at 100 Gbps line-rates, and this assumption can be violated by slow receivers. Backdraft addresses this problem and prevents congestion spreading from slow receivers.

9   Conclusions

In this paper, we present the design and implementation of Backdraft, a new lossless virtual switch. We make a case for providing lossless networking at the vswitch level by showing the impact of packet loss caused by slow receivers on network performance using existing congestion control algorithms.

We implemented Backdraft on top of the BESS virtual switch and performed experiments with two different clusters of servers on CloudLab (10 Gbps and 100 Gbps). We used unmodified POSIX applications with TAS TCP and a custom distributed application that performs RPCs with Homa, a state-of-the-art datacenter transport protocol. We demonstrate that Backdraft is effective in preventing packet loss and reduces tail latency by up to 20x compared to BESS.

References

[1] Af-xdp poll mode driver. https://doc.dpdk.org/guides/nics/af_xdp.html.

[2] Alveo SN1000 SmartNIC Accelerator Card. https://www.xilinx.com/products/boards-and-kits/alveo/sn1000.html.

[3] BESS: Berkeley Extensible Software Switch. https://github.com/NetSys/bess.

[4] Homa. https://github.com/PlatformLab/Homa.

[5] Homa commit 47265bf. https://github.com/PlatformLab/Homa/blob/392b577bbdad2f5aa42faefc88614992b5e505d2/src/TransportImpl.cc#L36.

[6] iperf3: Documentation. http://software.es.net/iperf/.

[7] Memcached. https://memcached.org/.

[8] Nginx. https://www.nginx.com/.

[9] Virtual Switch on BlueField SmartNIC. https://docs.mellanox.com/display/BlueFieldSWv20110841/Virtual+Switch+on+BlueField+SmartNIC.

[10] What's New in Virtio 1.1. https://www.dpdk.org/wp-content/uploads/sites/35/2018/09/virtio-1.1_v4.pdf.

[11] Xilinx alveo u280. https://www.xilinx.com/products/boards-and-kits/alveo/u280.html.

[12] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center tcp (dctcp). In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2010.

[13] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *Networked Systems Design and Implementation (NSDI)*. USENIX, 2012.

[14] N. Amit, A. Tai, and M. Wei. Don't shoot down tlb shootdowns! In *European Conference on Computer Systems (EuroSys)*. ACM, 2020.

[15] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*. ACM, 2012.

[16] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan. Attack of the killer microseconds. In *Communications of the ACM*, volume 60, pages 48–54. ACM, 2017.

[17] L. A. Barroso, J. Clidaras, and U. Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*. 2013.

[18] T. Benson, A. Akella, and D. A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2010.

[19] J. Bouron, S. Chevalley, B. Lepers, W. Zwaenepoel, R. Gouicem, J. Lawall, G. Muller, and J. Sopena. The battle of the schedulers: Freebsd ULE vs. linux CFS. In *Annual Technical Conference (ATC)*. USENIX, 2018.

[20] M. S. Brunella, G. Belocchi, M. Bonola, S. Pontarelli, G. Siracusano, G. Bianchi, A. Cammarano, A. Palumbo, L. Petrucci, and R. Bifulco. hXDP: Efficient software packet processing on FPGA nics. In *Operating Systems Design and Implementation (OSDI)*. USENIX, 2020.

[21] Q. Cai, S. Chaudhary, M. Midhul, Vuppalapati, J. Hwang, and R. Agarwal. Understanding Host Network Stack Overheads. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2021.

[22] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph. Understanding TCP incast throughput collapse in datacenter networks. In *Workshop on Research on Enterprise Networking (WREN)*. ACM, 2009.

[23] W. Cheng, K. Qian, W. Jiang, T. Zhang, and F. Ren. Re-architecting congestion management in lossless Ethernet. In *Networked Systems Design and Implementation (NSDI)*. USENIX, 2020.

[24] I. Cho, K. Jang, and D. Han. Credit-scheduled delay-bounded congestion control for datacenters. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2017.

[25] C. Chou, L. N. Bhuyan, and D. Wong. μdpm: Dynamic power management for the microsecond era. In *International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019.

[26] D. Crisan, R. Birke, N. Chrysos, C. Minkenberg, and M. Gusat. zFabric: How to virtualize lossless Ethernet? In *International Conference On Cluster Computing (CLUSTER)*. IEEE, 2014.

[27] D. Crisan, R. Birke, G. Cressier, C. Minkenberg, and M. Gusat. Got loss? get zOVN! In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2013.

[28] A. Daglis, M. Sutherland, and B. Falsafi. RPCValet: Ni-driven tail-aware balancing of µs-scale RPCs. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2019.

[29] W. J. Dally and C. L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. In *Transactions on Computers*, volume 36, pages 547–553. IEEE, 1987.

[30] M. Dalton, D. Schultz, J. Adriaens, A. Arefin, A. Gupta, B. Fahs, D. Rubinstein, E. C. Zermeno, E. Rubow, J. A. Docauer, et al. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *Networked Systems Design and Implementation (NSDI)*. USENIX, 2018.

[31] C. DeSanti. IEEE 802.1: 802.1Qbb - Priority-based Flow Control. http://www.ieee802.org/1/pages/802.1bb.html, 2009.

[32] D. Didona and W. Zwaenepoel. Size-aware sharding for improving tail latencies in in-memory key-value stores. In *Networked Systems Design and Implementation (NSDI)*. USENIX, 2019.

[33] C. Fang, H. Liu, M. Miao, J. Ye, L. Wang, W. Zhang, D. Kang, B. Lyv, P. Cheng, and J. Chen. Vtrace: Automatic diagnostic system for persistent packet loss in cloud-scale overlay network. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2020.

[34] A. Farshin, T. Barbette, A. Roozbeh, G. Q. Maguire Jr., and D. Kostić. PacketMill: Toward per-core 100-gbps networking. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2021.

[35] A. Farshin, A. Roozbeh, G. Q. M. Jr., and D. Kostić. Reexamining direct cache access to optimize i/o intensive applications for multi-hundred-gigabit networks. In *Annual Technical Conference (ATC)*. USENIX, 2020.

[36] A. Farshin, A. Roozbeh, G. Q. Maguire Jr, and D. Kostić. Reexamining direct cache access to optimize I/O intensive applications for multi-hundred-gigabit networks. In *Annual Technical Conference (ATC)*. USENIX, 2020.

[37] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *Special Interest Group on Programming Languages (SIGPLAN)*. ACM, 2012.

[38] D. Firestone. VFP: A virtual switch platform for host SDN in the public cloud. In *Networked Systems Design and Implementation (NSDI)*. USENIX, 2017.

[39] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg. Azure accelerated networking: SmartNICs in the public cloud. In *Networked Systems Design and Implementation (NSDI)*. USENIX, 2018.

[40] J. Fried, Z. Ruan, A. Ousterhout, and A. Belay. Caladan: Mitigating interference at microsecond timescales. In *Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, 2020.

[41] H. Golestani, A. Mirhosseini, and T. F. Wenisch. Software data planes: You can't always spin to win. In *Symposium on Cloud Computing (SoCC)*. ACM, 2019.

[42] R. Gouicem, D. Carver, J.-P. Lozi, J. Sopena, B. Lepers, W. Zwaenepoel, N. Palix, J. Lawall, and G. Muller. Fewer cores, more hertz: Leveraging high-frequency cores in the OS scheduler for improved application performance. In *Annual Technical Conference (ATC)*. USENIX, 2020.

[43] P. Goyal, P. Shah, K. Zhao, G. Nikolaidis, M. Alizadeh, and T. E. Anderson. Backpressure flow control. In *Networked Systems Design and Implementation (NSDI)*. USENIX, 2022.

[44] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn. RDMA over Commodity Ethernet at Scale. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2016.

[45] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy. SoftNIC: A software nic to augment hardware. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-155*, 2015.

[46] J. T. Humphries, K. Kaffes, D. Mazières, and C. Kozyrakis. Mind the gap: A case for informed request scheduling at the nic. In *Workshop on Hot Topics in Networks (HotNets)*. ACM, 2019.

[47] D. Intel. Data plane development kit, 2014.

[48] C. Iorgulescu, R. Azimi, Y. Kwon, S. Elnikety, M. Syamala, V. Narasayya, H. Herodotou, P. Tomita, A. Chen, J. Zhang, and J. Wang. Perfiso: Performance isolation for commercial latency-sensitive services. In *Annual Technical Conference (ATC)*. USENIX, 2018.

[49] K. Jang, J. Sherry, H. Ballani, and T. Moncaster. Silo: Predictable Message Latency in the Cloud. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2015.

[50] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, C. Kim, and A. Greenberg. EyeQ: Practical Network Performance Isolation at the Edge. In *Networked Systems Design and Implementation (NSDI)*. USENIX, 2013.

[51] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *Networked Systems Design and Implementation (NSDI)*. USENIX, 2019.

[52] K. Kaffes, J. T. Humphries, D. Mazières, and C. Kozyrakis. Syrup: User-defined scheduling across the stack. In *Symposium on Operating Systems Principles (SOSP)*. ACM, 2021.

[53] K. Kara, C. Hagleitner, D. Diamantopoulos, D. Syrivelis, and G. Alonso. High bandwidth memory on FPGAs: A data analytics perspective. In *International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2020.

[54] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez. Rubik: Fast analytical power management for latency-critical systems. In *International Symposium on Microarchitecture (MICRO)*. IEEE/ACM, 2015.

[55] G. P. Katsikas, T. Barbette, M. Chiesa, D. Kostić, and G. Q. Maguire. What you need to know about (smart) network interface cards. In *International Conference on Passive and Active Network Measurement*. Springer, 2021.

[56] A. Kaufmann, T. Stamler, S. Peter, N. K. Sharma, A. Krishnamurthy, and T. Anderson. TAS: TCP acceleration as an OS service. In *European Conference on Computer Systems (EuroSys)*. ACM, 2019.

[57] M. Kogias, G. Prekas, A. Ghosn, J. Fietz, and E. Bugnion. R2P2: Making RPCs first-class datacenter citizens. In *Annual Technical Conference (ATC)*. USENIX, 2019.

[58] A. Kopytov. Sysbench: a system performance benchmark. *http://sysbench. sourceforge. net/*, 2004.

[59] S. G. Kulkarni, W. Zhang, J. Hwang, S. Rajagopalan, K. K. Ramakrishnan, T. Wood, M. Arumaithurai, and X. Fu. NFVnice: Dynamic Backpressure and Scheduling for NFV Service Chains. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2017.

[60] G. Kumar, N. Dukkipati, K. Jang, H. M. G. Wassel, X. Wu, B. Montazeri, Y. Wang, K. Springborn, C. Alfeld, M. Ryan, D. Wetherall, and A. Vahdat. Swift: Delay is simple and effective for congestion control in the datacenter. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2020.

[61] P. Kumar, N. Dukkipati, N. Lewis, Y. Cui, Y. Wang, C. Li, V. Valancius, J. Adriaens, S. Gribble, N. Foster, and A. Vahdat. Picnic: Predictable virtualized nic. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2019.

[62] H. T. Kung, T. Blackwell, and A. Chapman. Credit-based flow control for ATM networks: Credit update protocol, adaptive credit allocation and statistical multiplexing. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 1994.

[63] B. Lepers, R. Gouicem, D. Carver, J.-P. Lozi, N. Palix, M.-V. Aponte, W. Zwaenepoel, J. Sopena, J. Lawall, and G. Muller. Provable multicore schedulers with ipanema: Application to work conservation. In *European Conference on Computer Systems (EuroSys)*. ACM, 2020.

[64] J. Leverich and C. Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *European Conference on Computer Systems (EuroSys)*. ACM, 2014.

[65] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *Symposium on Cloud Computing (SoCC)*. ACM, 2014.

[66] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: Improving resource efficiency at scale. In *International Symposium on Computer Architecture (ISCA)*. ACM, 2015.

[67] A. Manousis, R. A. Sharma, V. Sekar, and J. Sherry. Contention-aware performance prediction for virtualized network functions. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2020.

[68] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. Gribble, et al. Snap: a microkernel approach to host networking. In *Symposium on Operating Systems Principles (SOSP)*. ACM, 2019.

[69] A. Menon and W. Zwaenepoel. Optimizing TCP receive performance. In *Annual Technical Conference (ATC)*. USENIX, 2008.

[70] A. Mirhosseini, H. Golestani, and T. F. Wenisch. Hyperplane: A scalable low-latency notification accelerator for software data planes. In *International Symposium on Microarchitecture (MICRO)*. IEEE/ACM, 2020.

[71] A. Mirhosseini, B. L. West, G. W. Blake, and T. F. Wenisch. Q-zilla: A scheduling framework and core microarchitecture for tail-tolerant microservices. In *International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020.

[72] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2018.

[73] J. Ousterhout. A linux kernel implementation of the homa transport protocol. In *Annual Technical Conference (ATC)*. USENIX, 2021.

[74] H. Qin, Q. Li, J. Speiser, P. Kraft, and J. Ousterhout. Arachne: Core-aware thread management. In *Operating Systems Design and Implementation (OSDI)*. USENIX, 2018.

[75] R. Ricci, E. Eide, and The CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 2014.

[76] L. Rizzo. Netmap: a novel framework for fast packet i/o. In *Security Symposium (USENIX Security)*. USENIX, 2012.

[77] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the Social Network's (Datacenter) Network. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2015.

[78] R. Russell. virtio: towards a de-facto standard for virtual I/O devices. In *SIGOPS Operating Systems Review*, volume 42, pages 95–103. ACM, 2008.

[79] A. Saeed, N. Dukkipati, V. Valancius, V. The Lam, C. Contavalli, and A. Vahdat. Carousel: Scalable traffic shaping at end hosts. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2017.

[80] A. Saeed, Y. Zhao, N. Dukkipati, E. Zegura, M. Ammar, K. Harras, and A. Vahdat. Eiffel: Efficient and flexible software packet scheduling. In *Networked Systems Design and Implementation (NSDI)*. USENIX, 2019.

[81] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. Legoos: A disseminated, distributed OS for hardware resource disaggregation. In *Operating Systems Design and Implementation (OSDI)*. USENIX, 2018.

[82] E. Sharafzadeh, A. Sanaee, E. Asyabi, and M. Sharifi. Yawn: A cpu idle-state governor for datacenter applications. In *SIGOPS Asia-Pacific Workshop on Systems (APSys)*. ACM, 2019.

[83] A. Singhvi, A. Akella, D. Gibson, T. F. Wenisch, M. Wong-Chan, S. Clark, M. M. K. Martin, M. McLaren, P. Chandra, R. Cauble, H. M. G. Wassel, B. Montazeri, S. L. Sabato, J. Scherpelz, and A. Vahdat. 1RMA: Re-envisioning remote memory access for multi-tenant datacenters. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2020.

[84] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown. Programmable packet scheduling at line rate. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2016.

[85] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown. Programmable packet scheduling at line rate. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2016.

[86] A. Sriraman and A. Dhanotia. Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2020.

[87] B. Stephens, A. Akella, and M. Swift. Loom: Flexible and efficient NIC packet scheduling. In *Networked Systems Design and Implementation (NSDI)*. USENIX, 2019.

[88] B. Stephens, A. L. Cox, A. Singla, J. Carter, C. Dixon, and W. Felter. Practical DCB for improved data center networks. In *Conference on Computer Communications (INFOCOM)*. IEEE, 2014.

[89] A. Tootoonchian, A. Panda, C. Lan, M. Walls, K. Argyraki, S. Ratnasamy, and S. Shenker. ResQ: Enabling SLOs in network function virtualization. In *Networked Systems Design and Implementation (NSDI)*. USENIX, 2018.

[90] W. Tu, Y.-H. Wei, G. Antichi, and B. Pfaff. Revisiting the open vswitch dataplane ten years later. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2021.

[91] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller. Safe and effective fine-grained TCP retransmissions for datacenter communication. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2009.

[92] J. Yang, Y. Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *Operating Systems Design and Implementation (OSDI)*. USENIX, 2020.

[93] T. Yu, S. A. Noghabi, S. Raindel, H. Liu, J. Padhye, and V. Sekar. FreeFlow: High Performance Container Networking. In *Workshop on Hot Topics in Networks (HotNets)*. ACM, 2016.

[94] Z. Yu, C. Hu, J. Wu, X. Sun, V. Braverman, M. Chowdhury, Z. Liu, and X. Jin. Programmable packet scheduling with a single queue. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2021.

[95] Y. Yuan, M. Alian, Y. Wang, R. Wang, I. Kurakin, C. Tai, and N. S. Kim. Don't forget the I/O when allocating your LLC. In *International Symposium on Computer Architecture (ISCA)*. ACM, 2021.

[96] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz. DeTail: Reducing the flow completion time tail in datacenter networks. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2012.

[97] X. Zhan, R. Azimi, S. Kanev, D. Brooks, and S. Reda. Carb: A c-state power management arbiter for latency-critical workloads. In *Computer Architecture Letters*, volume 16, pages 6–9. IEEE, 2016.

[98] Y. Zhang, Y. Tan, B. Stephens, and M. Chowdhury. Justitia: Software multi-tenancy in hardware kernel-bypass networks. In *Networked Systems Design and Implementation (NSDI)*. USENIX, 2022.

[99] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang. Congestion control for large-scale RDMA deployments. In *Special Interest Group on Data Communication (SIGCOMM)*. ACM, 2015.
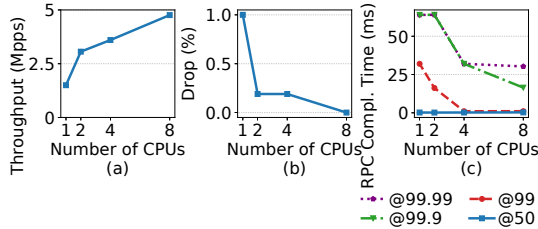
Figure 17: Throughput, loss and latency of DCTCP given different number of allocated cores. DCTCP still is unable to prevent packet loss with 4 CPU cores.
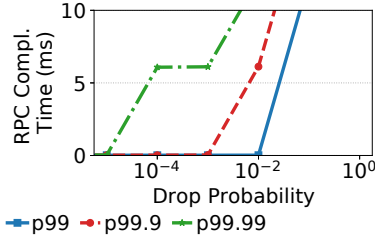

Figure 18: Homa experiences millisecond scale tail latency with even 10^{-2} drop probability.
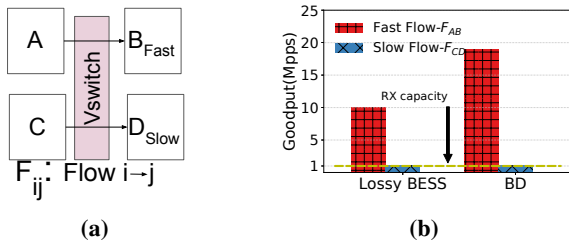

Figure 19: Backdraft TX bandwidth management in presence of a slow receiver. (a) The experiment setup on a single machine. (b) Backdraft prevents packet loss and saves CPU utilization from slow receiver and can allocate it to other receiver applications.

A Appendices

In this section, we expand our experiments associated with congestion controls, and bandwidth management on a single host.

A.1 Slow Receivers and DCTCP/Homa

While we discussed the problem associated with congestion controls such as Homa with regard to the slow receiver problem in Figure 4, we extended our study and performed similar experiments on DCTCP (§A.1.1).

We then discuss the impact of packet loss on latency of Homa (§A.1.2), given that Homa uses high granular timers to identify lost packets which is already discussed in §6.1.

A.1.1 DCTCP

We show that congestion control algorithms fail to address slow receiver problem in §2.1. Other than Homa, we per-

formed the same test on DCTCP congestion control. Figure 17a show that throughput of DCTCP application cannot reach higher than 5 Mpps or 320 Mbps with even 8 cores (64 B packets were used).

We have found that this packet loss occurs even when the vswitch performs ECN marking and end-hosts use a state-of-the-art congestion control algorithm like DCTCP [12]. This is demonstrated in Figure 17b, which shows what happens when we vary the number of allocated cores from 1 to 8 allocated to a DCTCP receiver application experiencing receiving data from a DCTCP client that is utilizing 8 CPU cores to send messages as fast as possible. We enable ECN marking at *vswitch level* to ensure DCTCP controls the flow rates in the scenarios where only vswitches are involved. Finally, Figure 17c demonstrates that packet loss has dramatic impact on the tail latency of the DCTCP.

A.1.2 Packet Loss Effect on Homa

In this section, we further discuss packet loss overhead of Homa protocol discussed in §4. In Figure 18, we observe that RPC completion time increase to 5x higher with mere packet loss probability of 10^{-2}. Although Homa identifies lost packets with high resolution timers, this does not seem to be highly effective.

A.2 Single Host Bandwidth Management

We performed an extra experiment to show how Backdraft works when dealing with a non-cooperative workload in terms of bandwidth management. This experiment is carried on a single node, we demonstrate that Backdraft delivers 2x higher throughput than its counterpart, BESS. Figure 19a shows the setup for this experiment. Here we have four applications (*A*, *B*, *C*, and *D*), where application *D* is a slow receiver and process packets at a maximum of 1 Mpps. The sender applications (i.e., *A* and *C*) are configured to transmit packets at 20 Mpps, instead. Reciver *B* is not limited in performance, so we can consider it to be fast. When Backdraft identifies the queue buildup due to slow receiver (i.e., *D*), it sends a local overlay message towards the sender port that includes a pause duration and an estimate of the receiver's rate. Using this information, Backdraft can pause the sender port, save CPU cycles otherwise wasted in handling the slow receiver flow, and use the saved resources to better handle the traffic directed to the fast receiver.

Figure 19b demonstrates this. With Backdraft, flow f_{AB} achieves 19 Mpps throughput. BESS, however, wastes CPU cycles and throughput bandwidth on dropping packets, causing the flow to reach only 10 Mpps.